

### Remarks

Applicants have amended claim 10 to more clearly define the present invention.

Examiner rejected claims 10-14 under 35 U.S.C. 112, first paragraph. In particular, the Examiner stated:

Claims 10-14 are rejected under 35 U.S.C. 112, first paragraph, as failing to comply with the enablement requirement. The claim(s) contains subject matter which was not described in the specification in such a way as to enable one skilled in the art to which it pertains, or with which it is most nearly connected, to make and/or use the invention. Applicant's exemplary claim 10 recites the limitation "determining a measure of CPU availability," however nowhere does Applicant's instant specification describe how such measure of CPU availability is determined. Applicant's specification merely recites in regards to the above limitation, "As described above other system metrics, such as CPU availability, may also be used." This exemplification provides no further description in such a way to enable one skilled in the art to make and/or use the invention. Furthermore, Examiner submits that it is unclear how a streaming media is capable of being played back to a client or how the client is able to receive such streaming media, without a CPU being available. Such lack of clarity convinces the Examiner that CPU availability is near obvious or inherent, however Examiner has utilized the broadest reasonable interpretation of "a measure of CPU availability" to mean the available a measure of which CPU (server) is actively available for processing. Therefore, Examiner suggest Applicant define the intended invention regarding the measure of CPU availability.

Applicants respectfully traverse this rejection.

Regarding claims 10-14: Applicants submit that the term CPU availability" is well known in the art. In particular, Applicants submit that one merely has to search for the term CPU availability, for example, using Google, and one will find numerous references to CPU availability" that were published prior to the earliest priority date. For example, see an article entitled "Predicting the CPU Availability of Time-shared Unix Systems" by R. Wolski, N. Spring and J. Hayes that was published in Proceedings of the Eighth International Symposium on High Performance Distributed Computing 1999, pp. 105-112, Meeting date 8/3/1999-8/6/1999 [Applicants have attached an early public version of the paper that was submitted on October 16, 1998]. Also, see a PowerPoint presentation entitled "Application Level Scheduling for Gene

Sequence Library Comparison for Metacomputing” by N. Spring and R. Wolski appearing in the ICS '98, July 15, 1998 (attached hereto) where slide 10 refers to CPU availability.

Applicants submit that from these references, as well as from the large number of references readily and publicly available on the Internet, the term “measure of CPU availability” was well known to those of ordinary skill in the art, and it was also well known to those of ordinary skill in the art how to determine such a measure. Further, Applicants submit that, as was well known to those of ordinary skill in the art, the term CPU availability is effectively the opposite of the term CPU utilization. See p.3 of an article entitled “Improving Processor Availability in the MPI Implementation for the ASCI/Red Supercomputer” by R. Brightwell, W. Lawry, A. Maccabe, and C. Wilson, Proceedings of the 27<sup>th</sup> Annual IEEE Conference on Local Computer Networks, 2002 (attached hereto) which states:

## 2.2 Availability and Utilization

The reader may note that we present our discussion and results in terms of processor *availability*, how much of the processor is available to the application, rather than processor *utilization*, how much of the processor is utilized during communication. The two terms are effectively inverses of one another, that is, high availability implies low utilization and vice versa. While it is common practice to report processor utilization, we find that utilization sends the wrong message. Utilization seems to be a good thing and one naturally assumes that higher utilization is better when, in fact, the opposite is actually the case.

Ultimately, the difference is one of perspective. We find that using the term availability keeps us focused on the fact that we are trying to provide resources to applications.

Referring to the specification, as captured in patent publication 2004/0064576, Applicants respectfully submit that the Examiner is incorrect in asserting that the specification does not adequately support the claims. In particular, Applicants refer to para. [0018] of the specification which states, in pertinent part:

... Thus, in accordance with the present invention, if a delay occurs during transmission of the audio or audio-visual work

from network 200 to US 300 (of course, it should be clear that such delays may result from any number of causes such as delays in accessing data from a storage device, delays in transmission of the data from a media server, delays in transmission through network 200, delays waiting for CPU resources on software implementations, and so forth), the playback rate is automatically slowed to reduce the amount of data drained from Capture Buffer 400 per unit time. As a result, and in accordance with the present invention, more time is provided for data to arrive at US 300 before the data in Capture Buffer 400 is exhausted. Advantageously, this delays the onset of data depletion in Capture Buffer 400 which would cause Playback System 500 to pause. (Emphasis added)

As the Examiner can readily appreciate from this, the specification has clearly indicated that issues related to CPU resources may be mitigated using time-scale modified playback rates. Further, para. [0024] states the following:

It should be understood that some embodiments of the present invention can operate in numerous modes. For example, one embodiment of the present invention may operate in a mode that attempts to balance a data consumption rate with a data arrival rate. In this mode, the embodiment utilizes changes in playback rate to alter the data consumption rate, and as a result, the playback rate of material presented by the embodiment is determined by the data delivery rate of information from the source, for example, a media server. For convenience, this mode is referred to as "Rate Determined by Data Arrival" mode. In another mode, an embodiment of the present invention: (a) monitors various system conditions and user input playback presentation rate requests; (b) computes or infers data arrival and departure rates; and (c) intervenes whenever a user request would cause data underflow or overflow in Capture Buffer 400 or a disruption in playback. For convenience, this mode is referred to as "Rate Restricted by Data Arrival" mode. (Emphasis added)

As the Examiner can readily appreciate, the underlined sections of para. [0024] refer to embodiments of the invention, and paras. [0122] and [0124] teach one of ordinary skill

how to utilize the measure of CPU availability in these two modes, which modes are described throughout the specification. In particular, para. [0122] states:

It should be understood that some embodiments of the present invention described above relate to presentation systems whose playback rates are determined by the media source transmitting data. Specifically, the media source, for example, a server, can elect to send data faster or slower than normal; to cause a faster or slower playback rate provided by these embodiments of User System 300. This mode was referred to above as the "Rate Determined by Data Arrival" mode. It should be understood that the data arrival rate is not the only metric which can be utilized to determine presentation or playback rate. As described above other system metrics, such as CPU availability, may also be used. (Emphasis added)

As the Examiner can appreciate from this, one of ordinary skill in the art is instructed how to utilize "other system metrics, such as CPU availability" in "these embodiments of User System 300." Similarly, as set forth in para. [0124]

Additionally, some embodiments of the present invention described above relate to presentation systems wherein a determination is made of maximum and minimum presentation rates that are allowable to provide continuous and uninterrupted playback of media existing locally on a storage device or transmitted from a remote storage device via a communication medium. In accordance with these embodiments, the maximum and minimum presentation rates may be used with other information to prevent users of a variable rate presentation system from specifying presentation rates (playback rates) that are outside ranges of rates for continuous and uninterrupted playback. This mode was referred to above as the "Rate Restricted by Data Arrival" mode. It should be understood that the data arrival rate is not the only metric which can be utilized to determine presentation or playback rate. As described above other system metrics, such as CPU availability, may also be used to prevent interruptions in playback. (Emphasis added)

As the Examiner can appreciate from this, one of ordinary skill in the art is instructed how “other system metrics, such as CPU availability, may also be used to prevent interruptions in playback” in embodiments of the “Rate Restricted by Data Arrival” mode.

As such, Applicants respectfully submit that the claims contain subject matter that was indeed described in the specification in such a way as to enable one skilled in the art to which it pertains, or with which it is most nearly connected, to make and/or use the invention.

Thus, in light of the above, claims 10-14 meet the requirements of 35 U.S.C. 112, first paragraph, and because of this, Applicants respectfully request the Examiner to withdraw this rejection.

Examiner rejected claims 10-14 and 18 under 35 U.S.C. 103(a). In particular, the Examiner stated:

Claims 10-14, and 18 are rejected under 35 U.S.C. 103(a) as being unpatentable over Gupta (US Publication No. 2002/0038374 A1) in view of Hoyer et al. (US Patent No. 6, 381, 635).

As per claim 10, Gupta teaches a method for playback of streaming media received over a non-deterministic delay network at a client device which comprises receiving the streaming media at the client device, which client device includes a CPU (figure 1 and col. 7, lines 38-50); playing back the streaming media; determining a time-scale modification rate considering one user input time-scale modification to prepare the streaming media for playback (col. 6, lines 39- 48, user input is used for timeline modification changes and rate for playback at the client device and col. 6, lines 63-col. 7, lines 1-3); and providing an indication of a current time-scale modification playback rate to the user (Figure 5, col. 10, lines 23-30).

Gupta does not explicitly teach determining a measure of CPU availability.

However Hoyer teaches determining a measure of CPU availability (col. 7, lines 10-21).

Accordingly, it would have been obvious to one of ordinary skill in the networking art at the time the invention was made to have incorporate Gupta's teachings to the teachings of Hoyer, for the purpose of routing request to client servers that are active and available (i.e. servers that have not failed or are in standby mode, col. 7, lines 10-21).

As per claim 11, Gupta-Hoyer teaches a method further comprises steps of providing an indication of a user requested time-scale modification playback rate (Figure 5, col. 10, lines 23- 30).

As per claim 12, Gupta-Hoyer teaches wherein the step of playing back comprises associating a time-scale modification playback rate with each entry in a playback buffer queue (col. 10, lines 53-62).

As per claim 13, Gupta-Hoyer teaches wherein the indication comprises a function of recent time-scale modification playback rates (col. 10, lines 53-62).

As per claim 14, Gupta-Hoyer teaches wherein the step of utilizing comprising ignoring or modifying the user input time-scale modification playback rate when it would interfere with providing continuous playback (col. 8, lines 40-44).

As per claim 18, Gupta teaches a method for playback of streaming media received over a non-deterministic delay network at a client device which comprises steps of: receiving the streaming media at the client device, which client device includes a CPU; playing back the streaming media; determining a time-scale modification playback rate as a function of the measure of CPU availability and utilizing time-scale modification to prepare the streaming media for playback.

Gupta does not teach determining a measure of CPU availability.

However, Hoyer teaches determining a measure of CPU availability (col. 7, lines 10-21).

Refer to the motivation of claim 10 which applies equally as well to claim 18.

Applicants have amended claim 10 to more clearly define the present invention.

As such, Applicants respectfully traverse this rejection.

**Discussion of Gupta:** Gupta teaches methods for streaming multimedia content over a network that supports user-specified timescale modification. See the Abstract which states: “Multimedia content is streamed over a network system from a server computer to a client computer. The client allows a user to enter a variable playback speed and varies the speed at which the multimedia content is rendered at the client. Time-scale modification technology is used to maintain the original pitch of any audio content, thereby maintaining its intelligibility.”

As set forth in para. [0009] in the Summary of the Invention:

The invention utilizes time-scale modification so that a user can vary the speed of streaming content without destroying its intelligibility. In accordance with the invention, a user selects multimedia content from a menu presented at a network client computer. In addition, the user selects a speed factor, indicating the speed at which the multimedia should be rendered relative to its default speed.

Further, at para. [0012], Gupta teaches how to deal with limited bandwidth situations:

The invention includes methods of adapting to limited bandwidth situations by composing or selecting composite streams having differing degrees of quality, and/or by composing or selecting streams with timelines that are altered to closely correspond with whatever speed factor has been chosen. In one embodiment of the invention, certain media streams, such as audio streams, take precedence over other streams such as video streams. In this embodiment of the invention, the audio stream is sent with an unaltered timeline, at a rate sufficient to satisfy the consumption requirements of the client, given the current speed factor. The video is then degraded in quality to reduce its bandwidth, so that it can be streamed in whatever bandwidth is not require by the audio.

Lastly, Gupta teaches the following with respect to interruptions in streaming at para. [0061]:

In the described embodiment, the user is allowed to change the speed designation during rendering of the composite media stream. In some cases, however, it may not be possible to change the playback speed without interrupting the playback momentarily. If this is the case, playback resumes as soon as possible, beginning at a point that shortly precedes the point at which playback was discontinued. Thus, there is some overlap in the presentation--when the presentation resumes, the overlap provides context for the new content that follows. (Emphasis added)

As the Examiner can readily appreciate from the above, Gupta teaches **no** action is to be taken to prevent or even mitigate the effects of emptying a data buffer. Because of this, Gupta leaves the user with interrupted playback. Hence, Gupta does not even address the problem solved by the invention of claims 10-14 and 18. In fact, as set forth above, Gupta only deals with a problem of limited bandwidth by preparing multiple streams having different degrees of quality. Applicants submit that Gupta's teaching in this regard is completely different from the inventions of claims 10-14 and 18.

In sum, Gupta does not teach, or suggest in any manner of how to deal with issues relating to CPU availability.

**Discussion of Hoyer:** As set forth in Hoyer at col. 2, lines 16-28:

The present invention greatly enhances the administrator's ability to analyze and understand a web site's performance by providing a display method that makes it easy to visualize the interrelationships between various performance measurements.

The present invention provides a compact mechanism for displaying multiple web performance graphs that allows the administrator to easily focus in on one or more performance measurements. The three key components of the present invention are 1) colored vertical scale buttons, 2) variable thickness colored graphs, and 3) colored legend buttons that can show or hide a particular graph.

As one can appreciate, Hoyer teaches nothing more regarding CPU utilization than to display it to a web administrator. In addition, Hoyer states the following regarding CPU utilization at col. 7, lines 52-55:

(3) CPU Utilization: is the number that represents the percentage of time that the CPU is doing useful work on a node running a web server. For web sites, it is the average of the node's CPU utilization numbers.

As the Examiner can readily appreciate, Hoyer does not teach or suggest using CPU utilization for any purpose whatsoever other than to display it.

Regarding claim 10: Applicants submit that there is no suggestion or motivation to combine the teachings of Gupta and Hoyer in any way whatsoever. In fact, Applicants respectfully submit that even if a person of ordinary skill in the art were to combine Gupta and Hoyer, that person would **not** arrive at the invention of claim 10 because there is nothing in either reference that teaches or hints in any way determining a time-scale modification playback rate considering a measure of CPU availability and user input time-scale modification playback rate requests as required by claims 10-14. In fact, if one of ordinary skill in the art were to combine Gupta and Hoyer, that person might display the CPU utilization or that person might generate alternative versions of the media being rendered that took less CPU utilization to present. Both of these clearly do not teach or suggest the invention of claim 10.



Regarding claim 11: Applicants respectfully submit that claim 11 depends from claim 10, and as such, is patentable over Gupta in view of Hoyer for the same reasons set forth above with respect to claim 10.

Regarding claim 12: Applicants respectfully submit that claim 12 depends from claim 10, and as such, is patentable over Gupta in view of Hoyer for the same reasons set forth above with respect to claim 10. In addition, Applicants submit that neither Gupta nor Hoyer teaches “wherein playing back comprises associating a time-scale modification playback rate with each entry in a playback buffer queue.” As such a combination of Gupta and Hoyer would not arrive at the invention of claim 12. Applicants respectfully submit that the Examiner is incorrect when she asserts: “Gupta-Hoyer teaches wherein the step of playing back comprises associating a time-scale modification playback rate with each entry in a playback buffer queue (col. 10, lines 53-62).”

As to claim 13: Applicants respectfully submit that claim 13 depends from claim 10, and as such, is patentable over Gupta in view of Hoyer for the same reasons set forth above with respect to claim 10.

As to claim 14: Applicants respectfully submit that claim 14 depends from claim 10, and as such, is patentable over Gupta in view of Hoyer for the same reasons set forth above with respect to claim 10. In addition, Applicants submit that neither Gupta nor Hoyer teaches “wherein the step of utilizing comprises ignoring or modifying the user input time-scale modification playback rate when it would interfere with providing continuous playback.” As such a combination of Gupta and Hoyer would not arrive at the invention of claim 14. Applicants respectfully submit that the Examiner is incorrect when she asserts: “**As per claim 14**, Gupta-Hoyer teaches wherein the step of utilizing comprising ignoring or modifying the user input time-scale modification playback rate when it would interfere with providing continuous playback (col. 8, lines 40-44).” In fact, Gupta teaches the opposite, i.e., using the user input leads to an interruption.

As to claim 18: Applicants submit that claim 18 is patentable over Gupta in view of Hoyer for substantially the same reasons set forth above with respect to claim 10.

Thus, in light of the above, Applicants respectfully submit that claims 10-14 and 18 are patentable over Gupta in view of Hoyer, and because of this, Applicants respectfully request the Examiner to withdraw this rejection.

In light of the above, Applicants respectfully submit that all the remaining claims are allowable, and Applicants respectfully request the Examiner to reconsider the case and pass the case to issue. Should the Examiner have any questions or wish to discuss any aspect of the application, a telephone call to the undersigned would be welcome.

Respectfully submitted,

By: /Michael B. Einschlag/  
Michael B. Einschlag  
Reg. No. 29,301  
(650) 949-2267  
25680 Fernhill Drive  
Los Altos Hills, California 94024

# Application Level Scheduling of Gene Sequence Library Comparison for Metacomputing

**Neil T. Spring and Rich Wolski**  
**University of California, San Diego**  
**ICS'98, July 15, 1998**

# Metacomputing

---

- **Almost all computational resources are networked**
  - Workstations, Supercomputers
  - Storage Devices
- **Increased network performance makes it possible to use distributed machines cooperatively**
- **How do you get performance? Scheduling.**

# AppLeS

---

## ■ Metacomputer Application Scheduling

- Assignment of tasks and data to resources
- Metacomputing environment
  - Distributed resources
  - Heterogeneity
  - Contention

## ■ Application Level Scheduling (AppLeS)

- Application specific control
- Resources are seen in terms of application-specific performance

# Gene Sequence Library Comparison

- Biological sequences (proteins, DNA) are compared using a fast, sequential heuristic
- Comparison based on biological criteria
  - Likelihood of various mutations
- Matches indicate related structure, function, origin

target library of sequences

source library

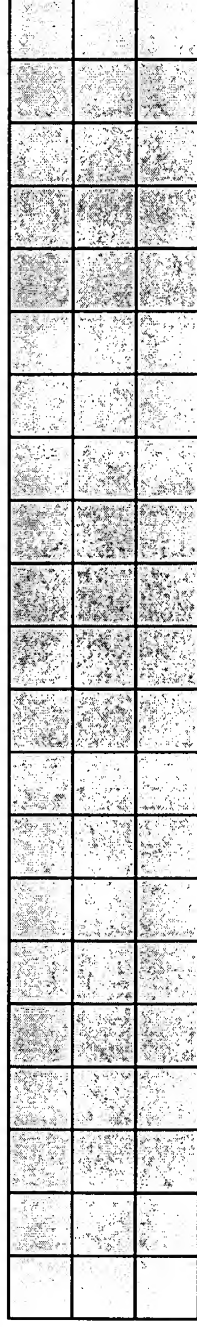
Results Array

# Parallel Library Comparison

- More memory, more processor cycles
- Sequence comparisons are independent
- Divide and Conquer approach
- Chunks of sequence libraries are distributed to different hosts and compared in parallel

target library of sequences

source library




Results Array

# Complib Implementation

---

## ■ Sequences compared using FASTA

- Sequence comparison heuristic
- Data dependent execution

## ■ Master/Slave structure:

- Master process distributes sequence library chunks
- Slave processes compute the solution and return results to the master



# Complib Scheduling Methods

---

## ■ Workstealing

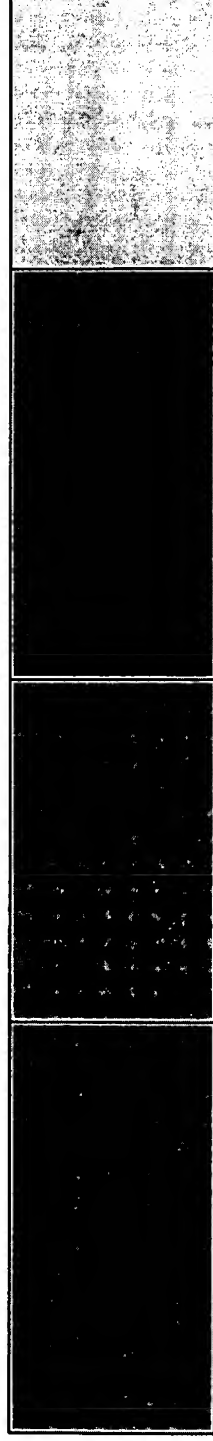
- Idle “slaves” request new work from “master”
- Size of work allocation determines load-imbalance
  - Tolerant of data dependent execution time
  - Tolerant of variance in processor performance

## ■ Static Placement

- Compute work placement before execution begins
- Low overhead
- Based on predicted execution time

# Run-time Static Placement

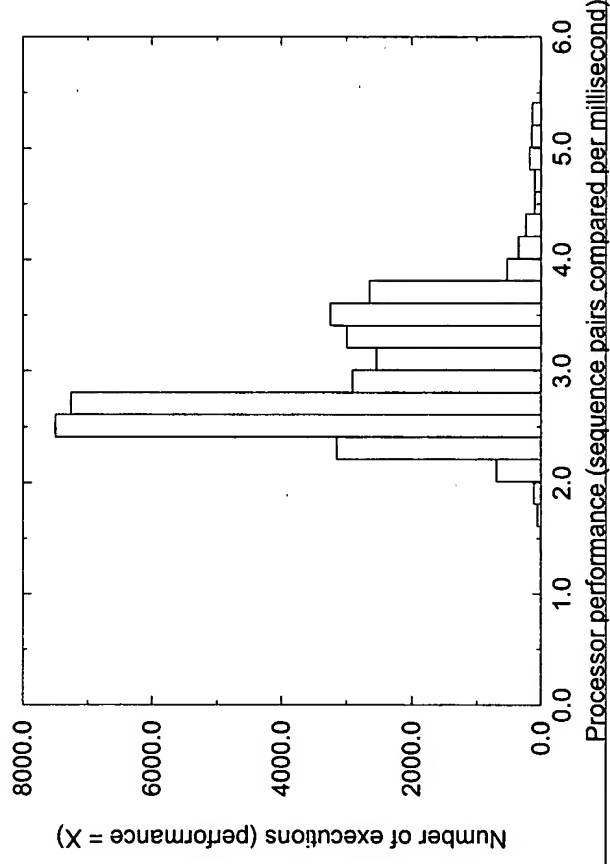
- **Time balancing: assign work so that all processors finish at the same time**
- **Execution time prediction for each slave**
  - Predict how long each task will take on each processor
  - Predict the “load” on each processor
- **Distribute the work in proportion to the relative capability of each processor**



AppLeS Static Strip Decomposition

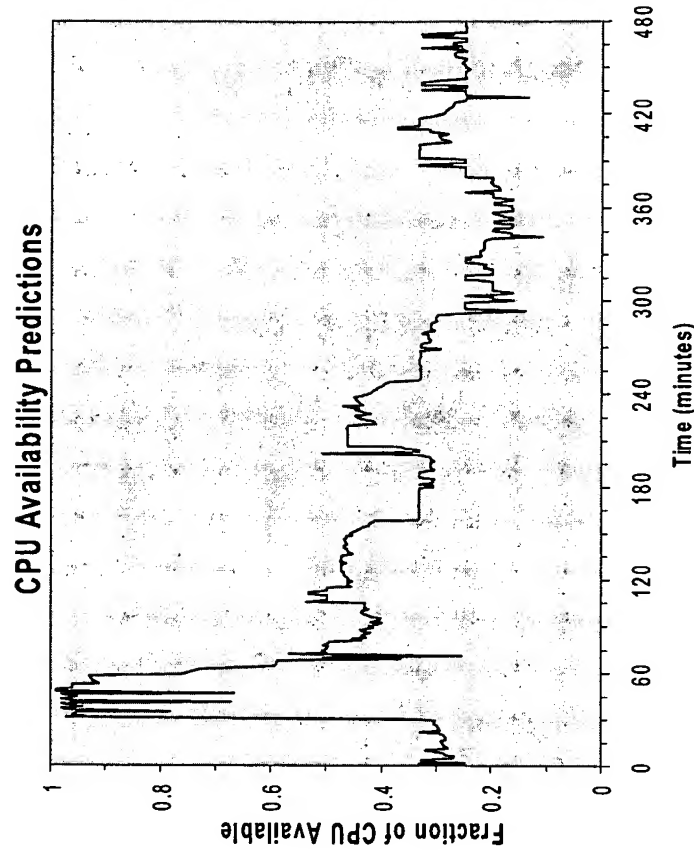
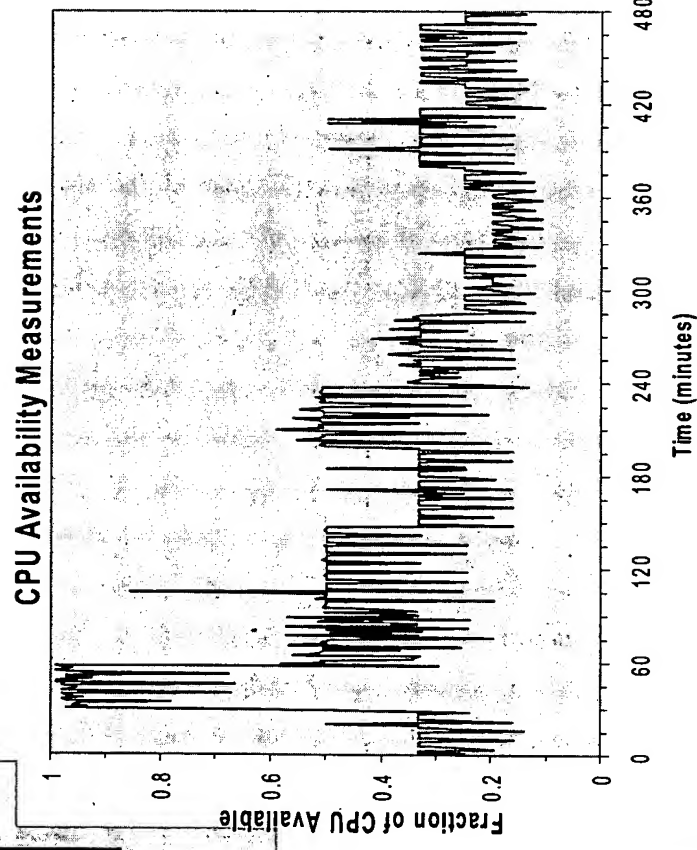
# Data Dependent Execution

- How long will it take to compare sequences even if there is no load?
- Data dependence ➡ Distribution of values
- Roughly symmetric ➡ Mean as representative



# Estimating load

- How much of the processor will we get?
- Use a time series of measurements
- To generate near term predictions



# **Problem with Run-time Static**

---

- **Point value estimates are not always accurate**
- **Variation in execution time from:**
  - Data dependent execution
  - Changes in contention
- **Causes load imbalance**
- **Potential to offset reduction of overhead over workstealing**

# Hybrid Scheduling Method

---

- **Reduce execution time using both:**
- **Placement (Run-time static)**
  - Master allocates work
  - Low communication & startup overhead
  - Place too much work ➡ load imbalance
- **Replacement (Workstealing)**
  - Slaves request additional work
  - Reduced load imbalance
  - Replace too much work ➡ excessive overhead
- **How much should be placed?**

# Hybrid Scheduling Boundary

---

## ■ Expected Performance

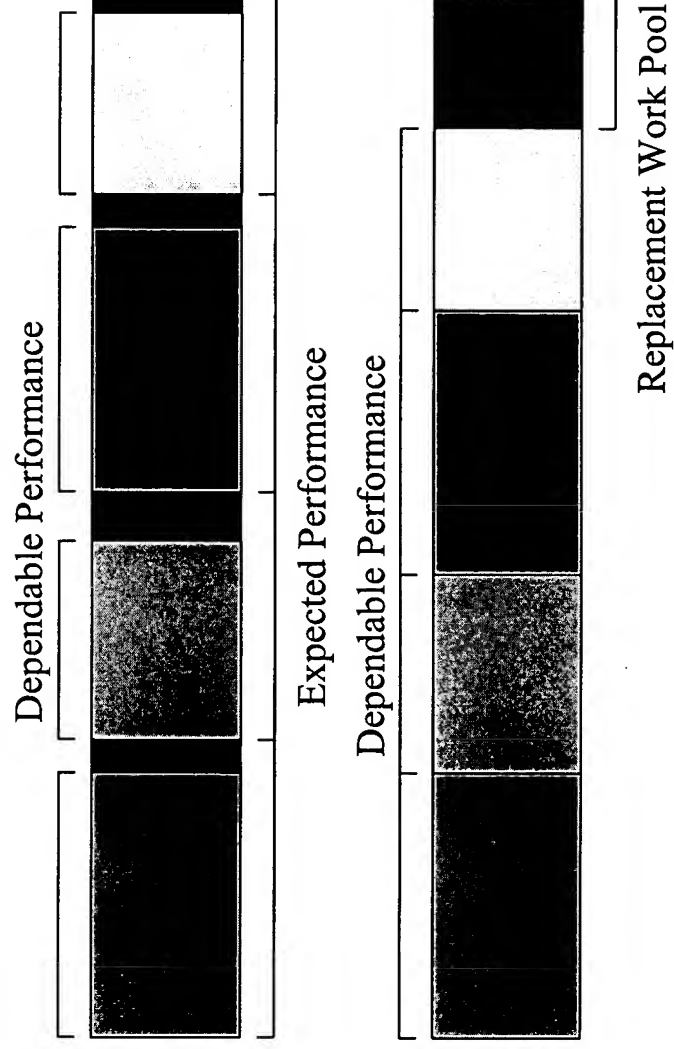
- Number of sequences we expect to be able to compare before the application finishes
- NWS Prediction & Application benchmark

## ■ Dependable Performance

- Artificial lower bound on performance
- NWS Prediction & Application benchmark & Variance

# Hybrid Scheduling

- Work partitioned based on *dependable* and *expected* performance
- Colors represent placed processor work, black represents work deferred for replacement





# Comparing Scheduling Methods

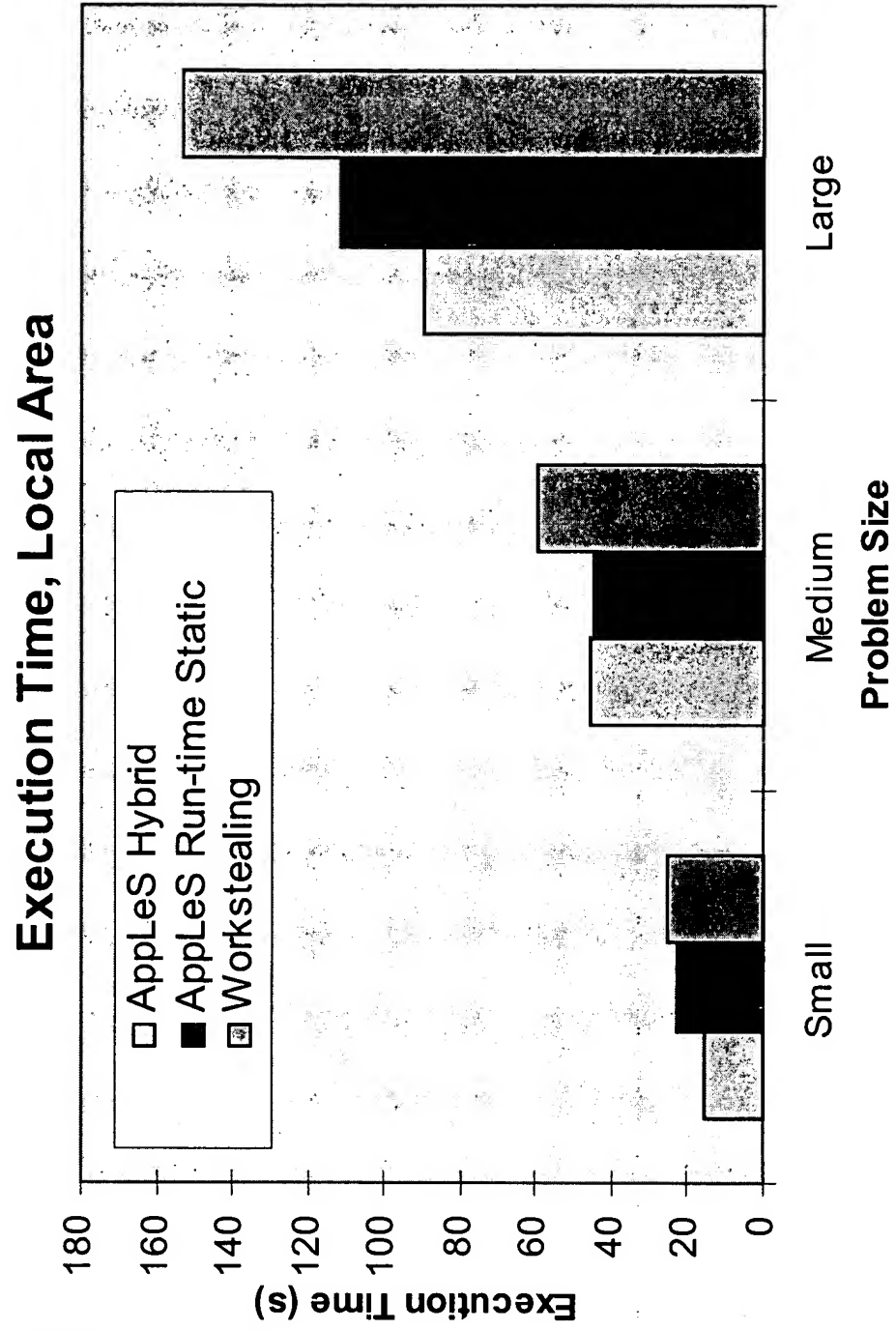
## ■ Three scheduling methods

- Workstealing: Replacement only
- Run-time Static: Placement only
- Hybrid: Placement & Replacement

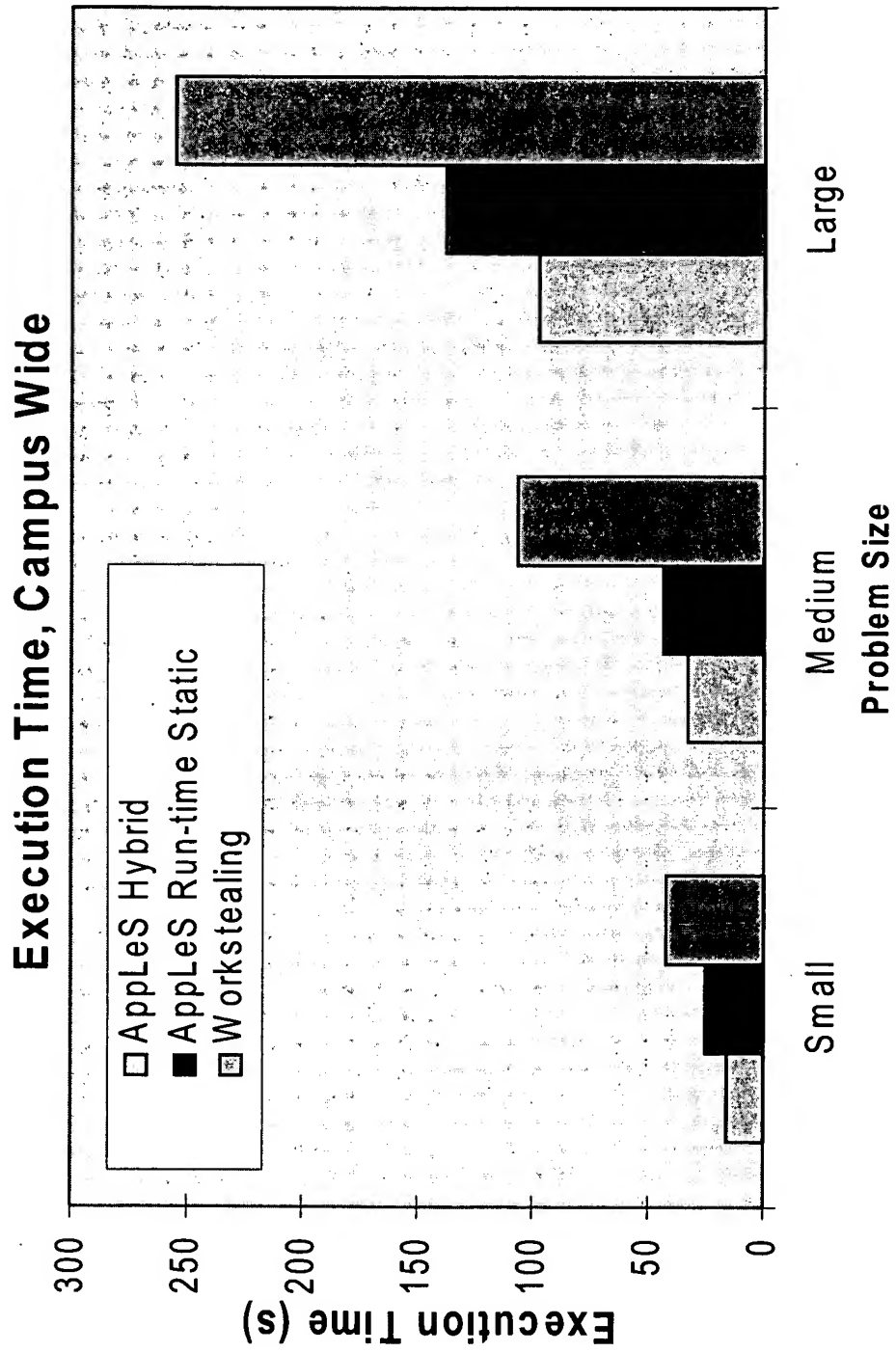
## ■ Three Platforms

- Local
  - SDSC HPC Cluster & 3 Workstations (25 processors)
- Campus
  - Adds PCL Suns (4) by campus backbone
- National
  - Adds CS6400 (12 processor) via cross-country ATM
  - And additional SDSC HPC machines (8 processors)

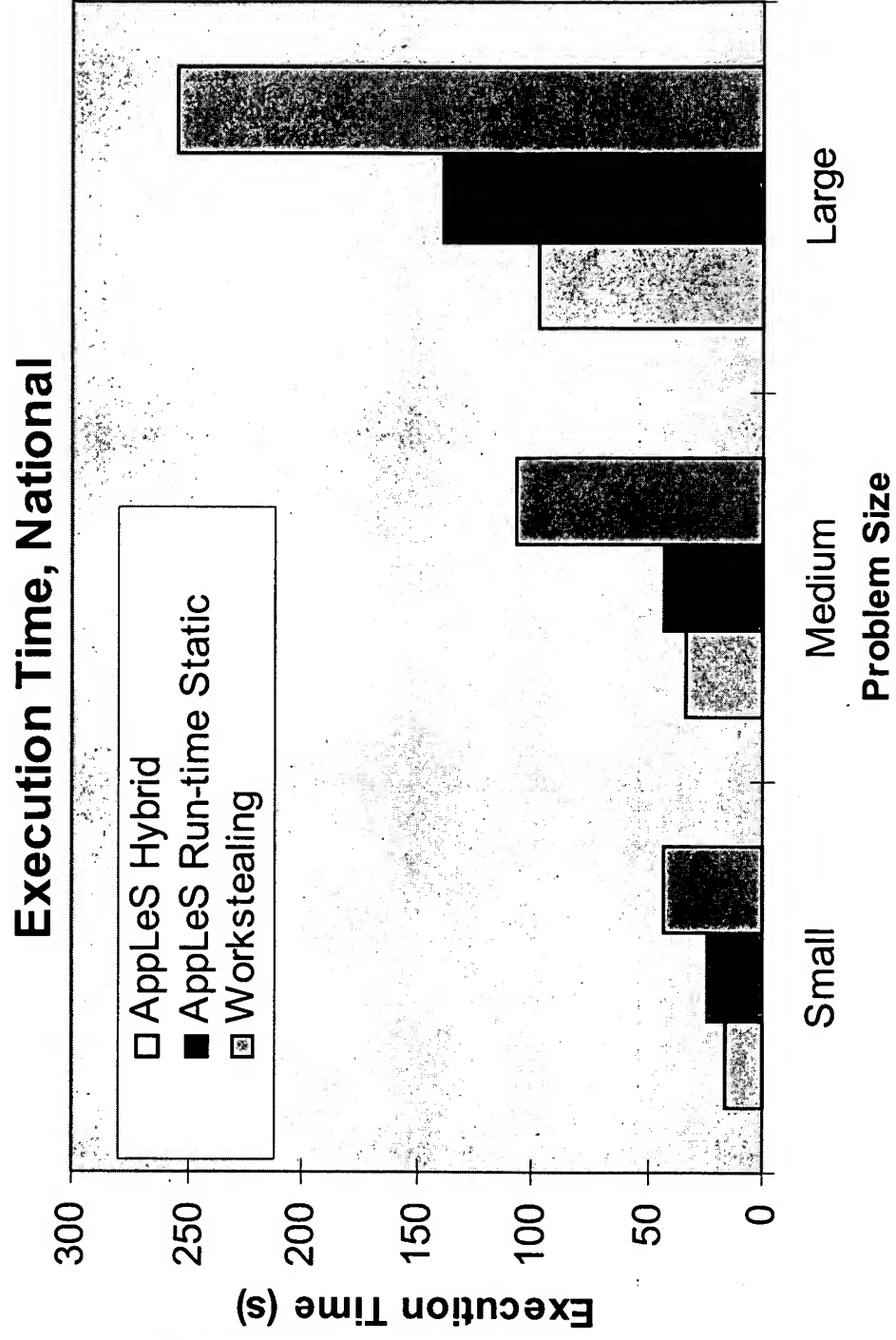
# Results: Local Area



# Results: Campus Wide



# Results: National



# Conclusions

---

- Possible to make predictions of data dependent execution time in a shared environment.
- Scheduling methods must tolerate inaccuracies necessarily part of predictive process
- Using predictions & accuracy measures we can combine scheduling policies for more efficient overall schedules

# Contact Information

---

■ **Neil Spring**

[nspring@cs.ucsd.edu](mailto:nspring@cs.ucsd.edu)

■ **Rich Wolski**

[rich@cs.ucsd.edu](mailto:rich@cs.ucsd.edu)

■ **AppLeS**

<http://www.cs.ucsd.edu/groups/hpcl/apples>

■ **Network Weather Service**

<http://nws.npaci.edu>

# Predicting the CPU Availability of Time-shared Unix Systems \*

UCSD Technical Report Number CS98-602  
(Submitted to SIGMETRICS '99)

Rich Wolski<sup>†</sup>, Neil Spring<sup>‡</sup>, Jim Hayes<sup>§</sup>

October 16, 1998

In this paper, we focus on the problem of making short and medium term forecasts of CPU availability on time-shared Unix systems. We evaluate the accuracy with which availability can be measured using Unix load average, the Unix utility `vmstat`, and the Network Weather Service CPU sensor that uses both. We also examine the autocorrelation between successive CPU measurements to determine their degree of self-similarity. While our observations show a long-range autocorrelation dependence, we demonstrate how this dependence manifests itself in the short and medium term predictability of the CPU resources in our study.

## 1 Introduction

Improvements in network technology have made the distributed execution of performance-starved applications feasible. High-bandwidth, low-latency local-area networks form the basis of low-cost distributed systems such as the HPVM [14], the Berkeley NOW [9], and various Beowulf [25] configurations. Similarly, common carrier support for high-performance wide-area networking is fueling an interest in aggregating geographically dispersed and independently managed computers.

---

\*Supported by NSF grant ASC-9701333 and Advanced Research Projects Agency/ITO under contract #N66001-97-C-8531.

<sup>†</sup>email: rich@cs.ucsd.edu

<sup>‡</sup>email: nspring@cs.washington.edu

<sup>§</sup>email: jhayes@cs.ucsd.edu

These large-scale *metacomputers* are capable of delivering greater execution performance to an application than is available at any one of their constituent sites [4]. Moreover, performance-oriented distributed software infrastructures such as Globus [12], Legion [18], Condor [26], NetSolve [7], and HPJava [6] are attempting to knit vast collections of machines into *computational grids* [13] from which compute cycles can be obtained in the way electrical power is obtained from an electrical power utility.

One vexing quality of these ensemble systems is that their performance characteristics vary dynamically. In particular, it is often economically infeasible to dedicate a large collection of machines and networks to a single application, particularly if those resources belong to separate organizations. Resources, therefore, must be shared, and the contention that results from this sharing causes the deliverable performance to vary over time. To make the best use of the resources that are at hand at any given point in time, an application scheduler (be it a program or a human being) must make a *prediction* of what performance will be available from each.

In this paper, we examine the problem of predicting available CPU performance on Unix times-shared systems for the purpose of building dynamic schedulers. In this vein, the contributions it makes are:

- an exposition of the *measurement error* associated with popular Unix load measurement facilities with respect to estimating the available CPU time a process can obtain,
- a study of one-step-ahead forecasting performance obtained by the Network Weather Service [29, 30] (a distributed, on-line performance forecasting system) when applied to CPU availability measurements,
- an analysis of this forecasting performance in terms of the autocorrelation present between consecutive measurements of CPU availability, and,



- verification of this analysis through its application to longer-range forecasting of CPU availability.

Our results are somewhat surprising in that they demonstrate the possibility of making short and medium range predictions of available CPU performance despite the presence of long-range autocorrelation and potential self-similarity. Since chaotic systems typically exhibit self-similar performance characteristics, self-similarity is often interpreted as an indication of unpredictability. The predictions we obtained, however, exhibited a mean absolute error of less than 10% typically making them accurate enough for use in dynamic process scheduling.

In the next section (Section 2), we detail the accuracy obtained by three performance measurement methodologies when applied to a collection of compute servers and workstations located in the U.C. San Diego Computer Science and Engineering Department. Section 3 briefly describes the Network Weather Service (NWS) [30, 29] – a distributed performance forecasting system designed for use by dynamic schedulers. The NWS treats measurement histories as time series and uses simple statistical techniques to make short-term forecasts. Section 3 also presents the analysis the forecasting errors generated by the NWS forecasting system for the resources we monitored in this study. Finally, we conclude with a discussion of these results in the context of dynamic scheduling for metacomputing and computational grid systems, and point to future research directions in Section 4.

## 2 Measurements and Measurement Error

For dynamic process scheduling, the goal of CPU prediction is to gauge the degree to which a process can be expected to occupy a processor over some fixed time interval. For example, if “50% of the CPU” is available from a time-shared CPU, a process should be able to obtain 50% of the time-slices over some time interval. Typically, the availability percentage is used as an expansion

factor [11, 23, 15, 1] to determine the potential execution time of a process. If only 50% of the time-slices are available, for example, a process is expected to take twice as long to execute as it would if the CPU were completely unloaded<sup>1</sup>. We have used *CPU availability* to successfully schedule parallel programs in shared distributed environments [24, 2]. Other scheduling systems such as Prophet [27], Winner [1], and MARS [15] use Unix load average to accomplish the same purpose.

In this section, we detail the error that we observed while measuring CPU availability. We report results gathered by monitoring a collection of workstations and compute servers in the Computer Science and Engineering Department at UCSD. We used the CPU monitoring facilities currently implemented as part of the Network Weather Service [30]. Each series of measurements spans a 24 hour period of “production” use during August of 1998. Despite the usual summertime hiatus from classwork, the machines in this study experienced a wide range of loads as many of the graduate students took the opportunity to devote themselves to research that had been neglected during the school year. As such, we believe that the data is a representative sample of the departmental load behavior.

## 2.1 Measurement Methods

For each system, we use the NWS CPU monitor to periodically generate three separate measurements of current CPU availability. The first is based on the Unix load average metric which is a one-minute smoothed average of run queue length. Almost all Unix systems gather and report load average values, although the smoothing factor and sampling rates vary across implementations. The NWS sensor uses the utility `uptime` to obtain a load average reading without special

---

<sup>1</sup>This relationship assumes that the execution interval is sufficiently large with respect to the length of a time-slice so that possible truncation and round-off errors are insignificant.

access privileges<sup>2</sup>. Using a load average measurement, we compute the available CPU percentage as

$$available\_cpu_{load\_avg} = \left( \frac{1.0}{Unix\_load\_avg + 1.0} \right) * 100\% \quad (1)$$

indicating the percentage of CPU time that would be available to a newly created process. Fearing load average to be insensitive to short-term load variability, we implemented a second measurement technique based on the utility `vmstat` which provides periodically updated readings of CPU idle time, consumed user time, and consumed system time, presented as percentages.

$$available\_cpu_{vmstat} = (idle\_time + \frac{user\_time}{rp + 1} + W * \frac{system\_time}{rp + 1}) \quad (2)$$

where  $rp$  is a smoothed average of the number of running processes over the previous set of measurements, and  $W$  is a weighting factor equal to  $user\_time$ . The rationale is that a process is entitled to the current idle time percentage, and a fair share of the user and system time percentages. However, if a machine is used as a network gateway (as was the case at one time in the UCSD CSE Department) user-level processes may be denied CPU time as the kernel services network-level packet interrupts. In our experience, the percentage of system time that is shared fairly is directly proportional to the percentage of user time, hence the  $W$  factor.

Lastly, we implemented a *hybrid* sensor that combines Unix load average and `vmstat` measurements with a small probe process. The probe process occupies the CPU for a short period of time (currently 1.5 seconds) and reports the ratio of the CPU time it used to the wall-clock time that passed as a measure of the availability it experienced. The hybrid runs its probe process much less frequently than it measures `available\_cpu_{load\_avg}` and `available\_cpu_{vmstat}` as these quantities may be obtained much less intrusively. The method (`vmstat` or load average) that reports

---

<sup>2</sup>The current implementation of the NWS runs completely without privileged access, both to reduce the possibility of breaching site security and to provide data that is representative of the performance an “average” user can obtain. For a more complete account of the portability and implementation issues associated with the NWS, see [30].

the CPU availability measure closest to that experienced by the probe is chosen to generate all measurements until the next probe is run. In the experiments described in this paper, the NWS hybrid sensor calculated *available\_cpu\_load\_avg* and *available\_cpu\_vmstat* every 10 seconds (6 times per minute), and probed once per minute. The method that was most accurate with respect to the probe was used for the subsequent 5 measurements each minute.

We are interested in the deliverable performance available for a full-priority Unix process that occupies the CPU for an appreciable amount of time. However, both Unix load average and the *vmstat* method are unable to sense the presence of lower priority or “nice” processes. It is our experience that users frequently run low-priority, background processes, particularly on shared resources to try and soak up any unused CPU time without arousing the ire of the departmental system administrators. To overcome this problem, the hybrid sensor uses the difference between the probe process and the most accurate method as a *bias* value and adjusts all subsequent measurements by this value. The assumption is that the probe will not be affected by lower-priority processes and hence will be able to bias the skewed measurements derived from the Unix load average or *vmstat*.

The advantage of using load average, *vmstat*, and the NWS-hybrid to derive measurements of CPU availability is that they are relatively non-intrusive. Both *vmstat* and *uptime* read protected Unix devices to access performance statistics maintained in the kernel. Presumably, these are not heavy-weight operations in most Unix implementations. Indeed, we notice little difference in CPU availability if two instances of either method are executing, which indicates that the load they generate is not measurable given their relative sensitivities. The NWS-hybrid, however, uses a short term spinning process which (in this study) executes once per minute. We have determined through experimentation that the shortest probe duration that is useful is 1.5 seconds. The overhead, then is 1.5/60 seconds or 2.5%.

## 2.2 Measurement Accuracy

To determine the accuracy of these three methods, we compare the readings they generate with the percentage of CPU cycles obtained by an independent ten-second, CPU-bound process which we will refer to as the *test process*. The *test process* executes and then reports the ratio of CPU time it received (obtained through the `getrusage()` system call) to total execution time (measured in wall-clock time) as the percentage of the CPU it was able to obtain. Measurement error, then, is defined as

$$\text{Measurement Error}_t = |\text{Measurement}_t - \text{Test Process Observation}_t| \quad (3)$$

where

$\text{Measurement}_t$  = measurement of CPU availability taken at time  $t$

$\text{Test Process Observation}_t$  = CPU availability observed by a *test process* at time  $t$

To avoid possible contention between the sensors and the test-process, we use the measurement taken most immediately before the *test process* executes as  $\text{Measurement}_t$ .

Table 1 details the measurement errors we observed for different hosts at UCSD. Each column shows the mean absolute difference between the CPU availability percentage quoted by the corresponding measurement method, and the availability percentage observed by the *test process* over a 24-hour, non-weekend period.

The hosts *thing1*, *thing2*, and *conundrum* are interactive workstations used for research by graduate students, while *beowulf*, *gremlin*, and *kongo* are general departmental servers available to faculty and students. Most of the errors are reasonably small and fairly equivalent across methods, given the dynamic nature of the systems we monitored. An error of 10% or less, for example, is considered useful for scheduling [23]. The notable exceptions are *conundrum* and

Host Name	Load Average	vmstat	NWS Hybrid
thing2	9.0%	11.2%	11.1%
thing1	6.4%	7.5%	6.1%
conundrum	34.1%	32.7%	4.4%
beowulf	6.3%	6.5%	7.5%
gremlin	4.0%	3.2%	4.1%
kongo	12.8%	12.9%	41.3%

Table 1: Mean Absolute Measurement Errors for UCSD Hosts during a 24-hour, mid-week period

*kongo*. On *conundrum*, a background process was running with Unix *nice* priority of 19 in an attempt to use otherwise unused CPU cycles. However, the *test process* runs with full priority, pre-empting the background process. The Unix load average and *vmstat* methods do not consider process priority, however, and record the system as being busy. The probe bias used by the NWS-hybrid method, however, correctly recognizes the priority difference and yields a more accurate measurement.

On *kongo* the NWS-hybrid performs dismally. During the monitor period, a long-running, full-priority process was executing on *kongo*. Typical Unix systems increase the rate at which process priority degrades while executing as a function of their CPU occupancy. A long-running process, therefore, will be temporarily evicted in favor of a short-running, full-priority process like the probe used by the NWS-hybrid sensor. The 1.5 second execution time of the probe is not long enough for it to contend with the long-running process, so the NWS-hybrid method does not sense its presence. The ten-second *test process*, however, executes long enough to share the processor with the resident long-running process and, consequently, receives a fraction better measured by both load average and *vmstat*. It is possible to increase the probe time of the NWS-hybrid sensor, with a corresponding increase in intrusiveness. We are working on other implementation techniques, however, to try to alleviate this problem.

For the purposes of predicting availability, the measurement error we observe serves as a upper bound on the accuracy of our forecasts. That is, we do not expect to forecast with greater accuracy

that we can measure. In general, at UCSD, the measurement errors we can obtain from these three methodologies are small enough so that measurements prove useful for scheduling. However, obtaining an accurate measure is complicated by the process priority mechanisms employed by Unix, and care must be taken when choosing a measurement methodology.

### 3 Forecasting

Forecasting, in this setting, is the prediction of the CPU availability that the *test process* will observe. We treat histories of measurements generated by each of the methods described in Section 2 as statistical time series. In this section, we discuss our methodology for using these time series to predict CPU availability, then compare the predictions generated with both subsequent measurements and subsequent *test process* observations to understand the error involved in the processes of prediction and forecasting. In Section 3.1 we discuss autoregressive and self-similar characteristics of these time series, and describe the effect of these characteristics on the accuracy of the predictions. In Section 3.2 we discuss the implications these characteristics have on predictions made for a longer time frame, and present additional results to show the increase in prediction error.

In previous work describing the NWS [29, 30, 31], we have proposed a methodology for making one-step-ahead predictions using computationally inexpensive time-series analysis techniques. Rather than use a single forecasting model, the NWS applies a collection of forecasting techniques to each series, and dynamically chooses the one that has been most accurate over the recent set of measurements. This method of dynamically identifying a forecasting model has been shown to yield forecasts that are equivalent to, or slightly better than, the best forecaster in the set [29]. To be efficient, each of the techniques must be relatively cheap to compute. We have borrowed heavily from methodologies used by the digital signal processing community [19] in our implementation. A complete description of each method and its relative advantages is provided in [29], [19], and

[16]. Briefly summarized, each method uses a “sliding window” over previous measurements to compute a one-step-ahead forecast based either on some estimate of the mean or median of those measurements.

To evaluate the accuracy of each forecast, we examine two forms of error. The first, given by Equation 4, compares a forecast for a specific time frame to the *test process* observation that is eventually made in that time frame. We term this form of error the *true forecasting error* as it represents the actual error a scheduler would observe. Note that, in the one-step-ahead case, the time at which the forecast is generated occurs immediately before the time frame in which the *test process* runs, hence the subscripts on the terms  $Forecast_{t-1}$  and  $Test Process Observation_t$  respectively. To distinguish the amount of error that results from measurement inaccuracy from error introduced by prediction, we also compute the *one step ahead prediction error* as given by Equation 5. This error represents the inaccuracy in predicting the next measurement that will be gathered in a particular series capturing the predictability of the series.

$$True\ Forecasting\ Error_t = |Forecast_{t-1} - Test\ Process\ Observation_t| \quad (4)$$

$$one\ step\ ahead\ prediction\ error = |Forecast_{t-1} - Measurement_t| \quad (5)$$

where

$$Forecast_{t-1} = \text{NWS forecast of CPU availability made at time } t - 1 \text{ for time } t$$

and  $Test\ Process\ Observation_t$  and  $Measurement_t$  are defined in Section 2.

Table 2 shows both the mean true forecasting error in boldface type and the mean measurement error (defined in Equation 3 and presented in Table 1) in parentheses across the various systems at UCSD. If the true forecasting errors and measurement errors are approximately the same, the process of predicting what the next measurement will be is not introducing much error. Table 3 illustrates this observation further. In it, we show the mean one-step-ahead prediction error, using



Host Name	Load Average	vmstat	NWS Hybrid
thing2	<b>8.9%</b> (9.0%)	<b>8.6%</b> (11.2%)	<b>10.3%</b> (11.1%)
thing1	<b>6.4%</b> (6.4%)	<b>7.0%</b> (7.5%)	<b>5.3%</b> (6.1%)
conundrum	<b>34.0%</b> (34.1%)	<b>32.6%</b> (32.7%)	<b>4.3%</b> (4.4%)
beowulf	<b>6.2%</b> (6.3%)	<b>6.8%</b> (6.5%)	<b>6.9%</b> (7.5%)
gremlin	<b>4.0%</b> (4.0%)	<b>2.6%</b> (3.2%)	<b>3.0%</b> (4.1%)
kongo	<b>12.8%</b> (12.8%)	<b>12.8%</b> (12.9%)	<b>41.0%</b> (41.3%)

Table 2: Mean **True Forecasting Errors** and Corresponding Measurement Errors (in parentheses) for UCSD Hosts during a 24-hour, mid-week period

Host Name	Load Average	vmstat	NWS Hybrid
thing2	1.2%	4.9%	1.8%
thing1	1.7%	3.1%	2.8%
conundrum	0.4%	0.2%	0.2%
beowulf	1.8%	3.1%	3.5%
gremlin	1.0%	2.1%	2.0%
kongo	0.1%	0.1%	0.1%

Table 3: Mean Absolute One-step-ahead Prediction Errors for UCSD Hosts during a 24-hour, mid-week period

the NWS forecasting techniques, for each measurement method on each of the systems that we studied. On each of these systems, the one-step-ahead prediction error is less than 5%. It is somewhat surprising that the one-step-ahead prediction error does not contribute more to the overall inaccuracy associated with predicting the *test process* values.

The instances in which forecast accuracy is better than measurement accuracy are curious. An analysis of the measurement and forecasting residuals is inconclusive with respect to the significance of this difference. Since the effect is generally small, however, we omit that analysis in favor of brevity and make the less precise observation that measurement and forecasting accuracy are approximately the same.

### 3.1 CPU Autocorrelation and Predictability

A plot of the autocorrelations as a function of previous lags reveals that CPU availability changes slowly with respect to time and hence can be predicted relatively accurately in the short term.

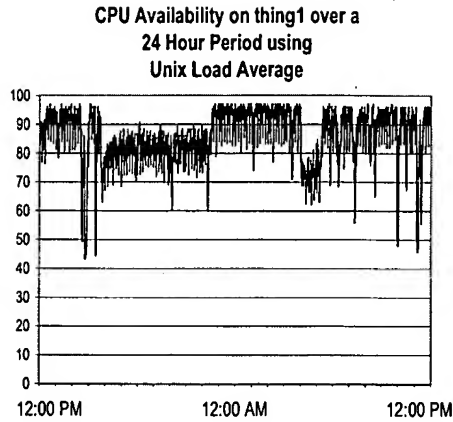


Figure 1: CPU Availability Measurements (using Unix Load Average) for *thing1*

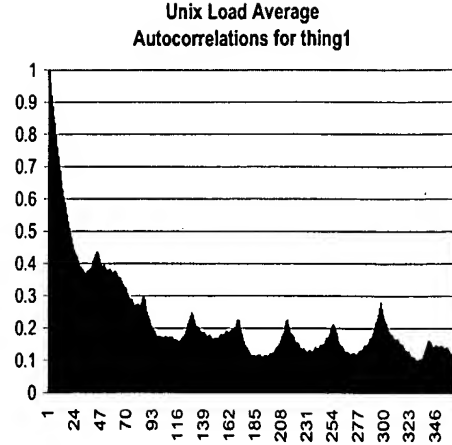


Figure 2: CPU Availability Autocorrelations (using Unix Load Average) for *thing1*

Figures 1 and 3 show time series plots of CPU availability measurements using Unix load average taken from *thing1* and *thing2* respectively. In Figures 2 and 4 we show the first 360 autocorrelations for Unix load average values taken from *thing1* and *thing2* respectively. In Figures 2 and 4 we show the first 360 autocorrelations for each series.

From both the time series and the plot of the autocorrelations, it is clear that events occurring even hours apart are correlated. However, the slow rate of decay in the autocorrelation function is suggestive of self-similarity, and self-similarity is often a manifestation of an unpredictable, chaotic series [5]. Recent studies of network packet traffic [20], World-Wide-Web traffic [8], network protocol performance [22], transmitted video traffic [3], and networked file systems [17] all point to self-similarity as an inherent property of modern distributed systems. Of particular interest is the work by Dinda and O'Halloran [10] in which the authors rigorously analyze Unix load average data from a large number of computational settings. The focus of their analysis is on the degree of self-similarity and long-range autocorrelation present in a set of traces taken from a large population of machines. In almost all cases, their work shows that CPU load (for the cases that they examined) is self-similar.

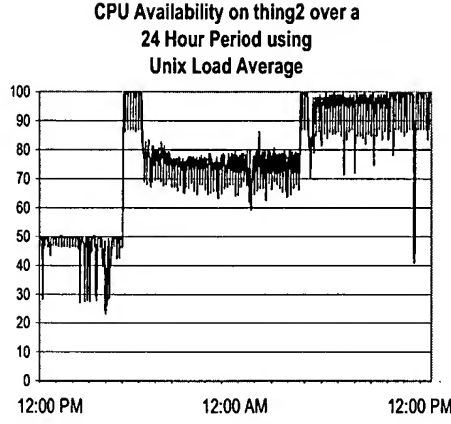


Figure 3: CPU Availability (using Unix Load Average) for *thing2*

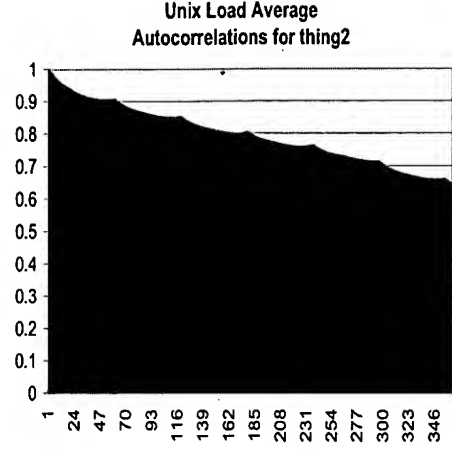


Figure 4: CPU Availability Autocorrelations (using Unix Load Average) for *thing2*

While there are several ways to characterize the degree of self-similarity in a series, the most common techniques estimate the Hurst parameter for the series. We defer to the work of Mandelbrot, Taqqu, Willinger, Leland, and Wilson [21, 20, 28], as well as the references cited in the previous paragraph, for a thorough exposition of the Hurst effect, its relationship to self-similarity, and various techniques for Hurst parameter estimation. For the purposes of determining self-similarity, though, it is enough to show that the Hurst parameter  $H$  of a series is likely to be between 0.5 and 1.0. To estimate this value for the data we gathered as part of this study, we use R/S analysis [21, 3] and pox plots [20] to determine if the Hurst parameter is likely to fall within this range. Briefly, for a given set of observations  $(X_k : k = 1, 2, \dots, n)$ , with sample mean  $\bar{X}(n)$  and sample variance  $\bar{S}^2(n)$ , the *rescaled adjusted range statistic* or R/S statistic is calculated as  $R(n)/S(n) = [\max(0, W_1, W_2, \dots, W_n) - \min(0, W_1, W_2, \dots, W_n)]/S(n)$  where  $W_k = (X_1 + X_2 + \dots + X_k) - k\bar{X}(n)$  ( $k \geq 0$ ). The expected value  $E[R(n)/S(n)] \sim cn^H$  for some constant  $c$  as  $n \rightarrow \infty$  where  $H$  is the Hurst parameter for the series. By partitioning the series of length  $N$  into non-overlapping segments of length  $d$ , and calculating  $R(d)/S(d)$  for each segment, as  $(1 \leq d \leq N)$  we obtain  $\lfloor d/N \rfloor$  samples of  $R(d)/S(d)$ . Plotting  $\log_{10}(R(d)/S(d))$

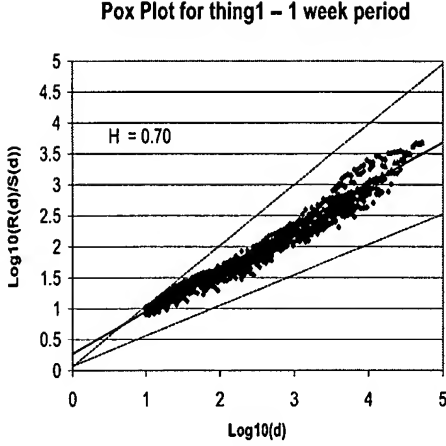


Figure 5: Pox Plot of CPU Availability using Unix Load average from *thing1*

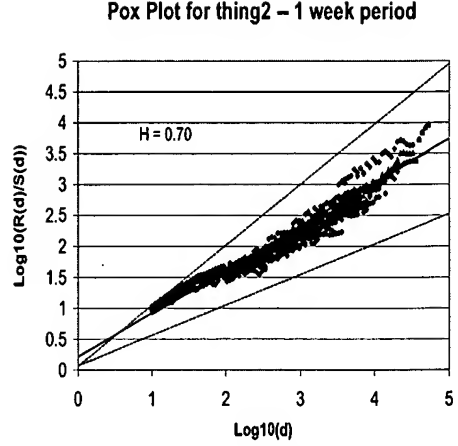


Figure 6: Pox Plot of CPU Availability using Unix Load average from *thing2*

versus  $\log_{10}(d)$  for each of these samples yields a pox plot for the series. Figures 5 and 6 show pox plots of CPU availability measurements using Unix load average for *thing1* and *thing2* over a one-week long period.

The two dotted lines in each figure depict slopes of 0.5 and 1.0. By inspection, any sort of “best fit” line for this data is likely to have a slope greater than 0.5 and less than 1.0, hence we can conclude that the Hurst parameter  $H$  falls somewhere in this range. In the figures, we show a least-squares regression line (solid) for the average  $\log_{10}(R(d)/S(d))$  value for each value of  $\log_{10}(d)$ . The slope of this line estimates the Hurst parameter as 0.70 for both *thing1* and *thing2* in the figure. In the second column of Table 4, we give the Hurst parameter estimations for each of the hosts in our study using this technique. The pervasiveness of these observations across our set of experiments supports the previous work of Dinda and O’Halloran. We, therefore, surmise that the CPU availability exhibits long-range autocorrelation and is either self-similar (as noted in [10]) or short-term self-similar as described in [17].

### 3.2 Prediction of CPU Availability Over Longer Time-frames

Despite the long-range autocorrelation present in CPU availability measurements, the data in Tables 2 and 3 show that one-step-ahead CPU availability is relatively predictable. The slowly decaying autocorrelation between measurements means that recent history is often a good predictor of the short-term future. That is, self-similarity does not imply short-term unpredictability.

Self-similarity does mean, however, that averaging values over successively larger time scales will not produce time series which are dramatically smoother. For a self-similar series  $X_1, X_2, \dots, X_n$ , with Hurst parameter  $H$ , and aggregation level  $m$ , the averaged series  $X^{(m)}$  has variance  $Var(X^{(m)}) \sim cm^{-\beta}$ ,  $H = 1 - \beta/2$  as  $m \rightarrow \infty$  where  $X^{(m)}$  is the series  $X_k^{(m)} = \frac{X_{km-m+1} \dots X_{km}}{m}$ ,  $k \geq 1$ . In other words, the variance of the average values of  $X^{(m)}$  decreases more slowly than the aggregation level  $m$  increases as  $m \rightarrow \infty$ .

It is an estimate of average CPU availability (and not a one-step-ahead prediction for the next 10 second time frame) that is most useful to a scheduler, as process execution time may be span minutes, hours, or days. By the relationship shown above, we would expect the variance associated with a prediction of the average availability over interval  $m$  to be no worse than that for a one-step-ahead prediction. Table 4 compares the variance of an aggregated series  $X^{(m)}$  where  $m$  corresponds to a five-minute interval with that of the original series for each of the hosts in our study. Except for *kongo* and the NWS hybrid sensor running on *conundrum*, the variance in each aggregated series is lower, as expected<sup>3</sup>.

Note that the decrease in variance resulting from aggregation does not necessarily imply that the aggregated series is more predictable. Table 5 shows the mean absolute one-step-ahead prediction error (Equation 4) for each of the aggregated series using the NWS forecasting methodology.

The one-step-ahead prediction for the aggregated series is typically less accurate than for the

---

<sup>3</sup>The *conundrum* case is curious as it appears self-similar, but the variance increases as the series is aggregated. We are studying this case to determine why this experiment defies the conventional analysis. The *kongo* value for the NWS hybrid sensor, however, is due to the leading constant  $c$  in the expression  $Var(X^{(m)}) \sim cm^{-\beta}$ . Additional aggregation of this series reveals a decreasing variance.

Host Name	Est. $H$	Load Average		vmstat		NWS Hybrid	
		orig.	300s	orig.	300s	orig.	300s
thing2	0.70	0.0348	<b>0.0338</b>	0.0431	<b>0.0351</b>	0.0321	<b>0.0315</b>
thing1	0.70	0.0081	<b>0.0062</b>	0.0103	<b>0.0048</b>	0.0147	<b>0.0090</b>
conundrum	0.79	0.0002	<b>0.0001</b>	0.0003	<b>0.0000</b>	0.0006	<b>0.0009</b>
beowulf	0.82	0.0058	<b>0.0039</b>	0.0063	<b>0.0019</b>	0.0151	<b>0.0057</b>
gremlin	0.71	0.0038	<b>0.0023</b>	0.0034	<b>0.0011</b>	0.0032	<b>0.0001</b>
kongo	0.69	0.0001	<b>0.0001</b>	0.0001	<b>0.0001</b>	0.0004	<b>0.0008</b>

Table 4: Variance of Original Series and 5 Minute Averages (in bold face)

Host Name	Load Average	vmstat	NWS Hybrid
thing2	<b>2.4%</b> (1.2%)	* <b>1.7%</b> (4.9%)	* <b>1.3%</b> (1.8%)
thing1	<b>4.9%</b> (1.7%)	<b>3.5%</b> (3.1%)	<b>3.9%</b> (2.8%)
conundrum	<b>0.7%</b> (0.4%)	<b>0.2%</b> (0.2%)	<b>0.3%</b> (0.2%)
beowulf	<b>3.4%</b> (1.8%)	* <b>2.3%</b> (3.1%)	<b>4.5%</b> (3.5%)
gremlin	<b>2.6%</b> (1.0%)	* <b>1.2%</b> (2.1%)	* <b>1.3%</b> (2.0%)
kongo	<b>0.2%</b> (0.1%)	<b>0.1%</b> (0.1%)	<b>0.2%</b> (0.1%)

Table 5: Mean Absolute One-step-ahead Prediction Errors for 5 Minutes Aggregated UCSD Hosts during a 24-hour, mid-week period. Unaggregated error, from Table 3, is parenthesized.

original series. For the cases denoted by a \* in the table, however, the aggregated prediction is more accurate than the corresponding one-step-ahead, 10 second, prediction. We hypothesize that smoothing may be more effective for certain time frames (aggregation levels) than for others. The prediction error, therefore, may improve for these aggregation levels, and the smoothed series may be predicted more accurately. This hypothesis supports the similar observations made in [10] and [17] regarding the smoothness of aggregated series. In general, however, the improvement should be small and there is no trend as a function of aggregation level that we can detect.

To gauge the true forecasting error in the aggregated case, we examine the difference between the forecasted value and the value observed by a *test process*. This new *test process* runs for 5 minutes at a time, every 60 minutes. The new forecasted value is derived from the averaged series  $X^{30}$ , which is calculated as  $X_t^{30} = \frac{X_{30t-30+1} \dots X_{30t}}{30}$  for  $t$  varying from 1 to the number of entries in each trace, counting by 30. Since we obtain a measurement every 10 seconds, each  $X^{30}$

Host Name	Load Average	vmstat	NWS Hybrid
thing2	6.6%	5.3%	6.5%
thing1	5.6%	5.2%	6.7%
conundrum	3.0%	7.4%	10.1%
beowulf	6.0%	11.4%	11.1%
gremlin	4.3%	2.9%	8.3%
kongo	2.1%	1.9%	28.5%

Table 6: Mean True Forecasting Errors for 5 Minute Average CPU Availability on UCSD Hosts during a 24-hour, mid-week period

value is an average the measurements taken over five minutes. We then consider a one-step-ahead forecast of each  $X^{30}$  value as a prediction of the average availability during the succeeding five minute period. To calculate the aggregate true forecasting error, shown in Table 6, we again use Equation 5, where  $t$  now represents 5 minutes. Again, a problem with the bias value used by the NWS hybrid sensor causes the large discrepancy on kongo.

Note that we execute the *test process* only once every 60 minutes to prevent the load induced by them from driving away potential contention. We feared that a more frequent execution of the *test process* might cause other users to abandon the hosts we were monitoring in favor of more lightly loaded alternatives.

Figures 7 and 8 show the Unix load average CPU availability measurements during the 24-hour experimental period for hosts *thing1* and *thing2* respectively as example traces. From these figures, it is clear that the systems experienced load during the test period (the apparent periodic signal results from the intrusiveness of the 5 minute *test process*). Despite the variance in each series, however, the average true forecasting error for a program that occupies the CPU for five minutes is between 5% and 6%.

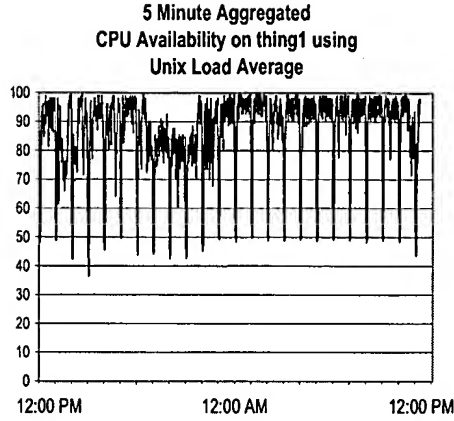


Figure 7: 5 Minute Aggregated CPU Availability using Unix Load average from *thing1*

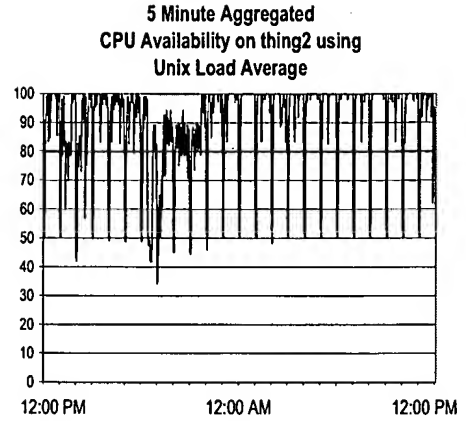


Figure 8: 5 Minute Aggregated CPU Availability using Unix Load average from *thing2*

## 4 Conclusions, Ramifications, and Future Work

From the data presented in Sections 2 and 3 we make the following observations about the workstations and computational servers in the UCSD Computer Science and Engineering department during the experimental period:

- Using conventional, non-privileged, Unix utilities the greatest source of error in making a one-step-ahead prediction of CPU availability comes from the process of *measuring* the availability of the CPU and not from *predicting* what the next measurement value will be.
- Traces of CPU availability exhibit long-range autocorrelation structures and are potentially self-similar.
- Short-term (10 seconds) and medium-term (5 minute) predictions of CPU availability (including all forms of error) can be obtained that are, on the average, between 5% and 12%.



In the context of process scheduling, these results are encouraging and somewhat surprising. Often, researchers assume that CPU loads vary to such a degree as to make dynamic scheduling difficult or impossible. While we certainly observe variation which is sometimes large, the series that are generated are fairly predictable. Moreover, the measurement and forecast error combined are small enough so that effective scheduling is possible. In [24], for example, we used considerably less accurate measurements to achieve performance gains that were better than 100% in some cases.

Another important realization is that long-range autocorrelation and self-similarity do not necessarily imply short-term unpredictability. Much of the previous excellent analysis work has focused on identifying and explaining self-similar performance behavior, particularly of networks. In these domains, short-term predictability may not be as important as predicting the long-term. While it is true that long-term predictions would be useful in a process scheduling context, short-term predictability also has utility.

Lastly, our observations coincide with those made recently by Dinda and O'Halloran [10] with respect to observed autocorrelation structure and Unix load average measurements. This is fortuitous since several large-scale metacomputing systems [18, 12, 26] use Unix load average to perceive system load. We extend this previous work by attempting to quantify the measurement error inherent in using load average as a measure of CPU availability, and by quantifying the effectiveness of the current NWS forecasting techniques on this type of data.

In future studies, we wish to expand the types of resources we consider to shared-memory multiprocessors, and collections of workstations that are combined using specialized networks (e.g. the Berkeley NOW [9]). We will also extend our set of experimental subjects to include workstations and computational servers in different production environments.

## References

- [1] O. Arndt, F. Bernd, T. Kielmann, and F. Thilo. Scheduling parallel applications in networks of mixed uniprocessor/multiprocessor workstations. In *Proceedings of ISCA 11th Conference on Parallel and Distributed Computing*, September 1998.
- [2] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing 1996*, 1996.
- [3] M. Bernan, R. Serman, and M. Taqu. Long-range dependence in variable-bit-rate video traffic. *IEEE Transactions on Communications*, March 1995.
- [4] S. Burnett and S. Fitzgerald. Metacomputing supports large-scale distributed simulations, 1998. available from <http://www.cacr.caltech.edu/sfexpress/sc98.html>.
- [5] A. Cambel. *Applied Chaos Theory*. Academic Press, 1993.
- [6] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. Hpjava: Data parallel extensions to java. In *ACM Workshop on Java for High-Performance Network Computing*, 1998. available from <http://www.cs.ucsb.edu/conferences/java98/papers/hpjava.pdf>.
- [7] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. of Supercomputing'96, Pittsburgh*. Department of Computer Science, University of Tennessee, Knoxville, 1996.
- [8] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5, December 1997.
- [9] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel computing on the berkeley now. In *to appear in JSPP'97 (9th Joint Symposium on Parallel Processing)*, 1997. available from <http://now.CS.Berkeley.EDU/Papers2>.
- [10] P. Dinda and D. O'Halloran. The statistical properties of host load. In *to appear in the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98) and CMU Tech. report CMU-CS-98-143*, 1998. available from <http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU-CS-98-143.ps>.
- [11] S. Figueira and F. Berman. Modeling the effects of contention on the performance of heterogeneous applications. In *Proc. 6th IEEE Symp. on High Performance Distributed Computing*, August 1997.
- [12] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997. to appear.
- [13] I. Foster and C. Kesselman, editors. *The Grid - Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
- [14] B. Ganguly, G. Bryan, M. Norman, and A. Chien. Exploring structured mesh refinement (samr) methods with the illinois concert system. In *Proceedings of SIAM conference on Parallel Processing*, March 1997.
- [15] J. Gehring and A. Reinfeld. Mars - a framework for minimizing the job execution time in a metacomputing environment. *Proceedings of Future general Computer Systems*, 1996.
- [16] C. Granger and P. Newbold. *Forecasting Economic Time Series*. Academic Press, 1986.
- [17] S. Gribble, S. Manku, D. Roselli, and E. Brewer. Self-similarity in file systems. available from <http://www.cs.berkeley.edu/~gribble>.
- [18] A. S. Grimshaw, W. A. Wulf, J. C. French, A. C. Weaver, and P. F. Reynolds. Legion: The next logical step toward a nationwide virtual computer. Technical Report CS-94-21, University of Virginia, 1994.
- [19] R. Haddad and T. Parsons. *Digital Signal Processing: Theory, Applications, and Hardware*. Computer Science Press, 1991.
- [20] W. Leland, M. Taqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, February 1994.

- [21] B. B. Mandelbrot and M. S. Taqqu. Robust R/S Analysis of Long-run serial Correlation. In *Proceedings of the 42nd Session of the ISI*, volume 48, pages 69–99, 1979.
- [22] K. Park, G. Kim, and M. Crovella. On the effect of traffic self-similarity on network performance. In *Proceedings of the SPIE International Conference on Performance and Control of Network Systems*, November 1997.
- [23] J. Schopf and F. Berman. Performance prediction in production environments. In *Proceedings of IPPS/SPDP 1998*, 1998. available from <http://www.cs.ucsd.edu/users/jenny/TechReports/ipps98.ps>.
- [24] N. Spring and R. Wolski. Application level scheduling: Gene sequence library comparison. In *Proceedings of ACM International Conference on Supercomputing 1998*, July 1998.
- [25] T. Sterling, D. Becker, and D. Savarese. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the International Conference on Parallel Processing*, 1995.
- [26] T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobbs Journal*, February 1995.
- [27] J. Weissman and A. Grimshaw. A framework for partitioning parallel computations in heterogeneous environments. *Concurrency: Practice and Experience*, 7(5), August 1995.
- [28] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-similarity through high-variability: statistical analysis of ethernet lan traffic at the source level. In *SIGCOMM'95 Conference on Communication Architectures, Protocols, and Applications*, 1995.
- [29] R. Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1998. also available from <http://www.cs.ucsd.edu/users/rich/publications.html>.
- [30] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems (to appear)*, 1998. available from <http://www.cs.ucsd.edu/users/rich/papers/nws-arch.ps>.
- [31] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Proceedings of Supercomputing 1997*, November 1997.

# Improving Processor Availability in the MPI Implementation for the ASCI/Red Supercomputer\*

Ron Brightwell, William Lawry  
Scalable Computing Systems Department  
Sandia National Laboratories<sup>†</sup>  
P.O. Box 5800  
Albuquerque, NM, 87111-1110  
{bright,wflawry}@cs.sandia.gov

Arthur. B. Maccabe, Christopher Wilson  
Scalable Systems Lab  
Computer Science Department  
University of New Mexico  
Albuquerque, NM, 87131-1386  
{maccabe,riley}@cs.unm.edu.

## Abstract

*This paper describes how a portable benchmark suite that measures the ability of an MPI implementation to overlap computation and communication can be used to discover and diagnose performance problems. We describe the approach of the benchmark suite and discuss a performance problem that we uncovered with the MPI implementation on the ASCI/Red supercomputer. A slight modification to the MPI implementation has resulted in a significant gain CPU availability and bandwidth with a slight degradation in latency performance. We present a detailed analysis of these results and discuss how the benchmark suite has enabled us to tailor the MPI implementation to optimize for all three measurements.*

**Keywords:** System-area network, Message-passing, MPI, Performance Analysis

## 1 Introduction

We have designed and implemented a portable benchmark suite called COMB, the Communication Offload MPI-based Benchmark, that measures the ability of an MPI implementation to overlap computation and MPI communication. The ability to overlap computation with communication is influenced by several system charac-

teristics, such as the quality of the MPI implementation and the capabilities of the underlying network transport layer. For example, some message passing systems interrupt the host CPU to obtain resources from the operating system in order to receive packets from the network. This strategy is likely to adversely impact the utilization of the host CPU, but may allow for an increase in MPI bandwidth.

While our benchmark was developed to measure the quality of MPI implementations on clusters, during the initial development of the benchmark suite, we made several runs on the ASCI/Red supercomputer at Sandia National Laboratories. Initially the runs were made to validate the results of the benchmark suite on a tightly-coupled parallel platform. However, the benchmark suite revealed a subtle but significant performance problem with the MPI implementation. In relating these results and a subsequent change to the implementation to correct the problem, we demonstrate that the benchmark suite can provide greater insight into the relationship between network performance and CPU performance.

The rest of this paper is organized as follows. In the next section, we provide general background for our interest in processor availability. Section 3 describes the benchmark suite. In Section 4, we provide an overview of the hardware and software environment of the ASCI/Red supercomputer. Section 5 presents initial results from the benchmark suite and describes the performance problem that was revealed. This section continues by describing a small MPI enhancement and the resulting performance impact. Section 6 discusses the conclusions of this paper.

\*This work was supported in part through the Computer Science Research Institute (CSRI) at Sandia National Laboratories under contract number SF-6432-CR.

<sup>†</sup>Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

<b>Producer:</b> double A[BSIZE], B[BSIZE];  fill A; wait CTS A; isend A;  fill B; wait CTS B; isend B;  for( i = 0 ; i < n-1 ; i++ ) { wait A sent; fill A; wait CTS A; isend A;  wait B sent; fill B; wait CTS B; isend B; }	<b>Consumer:</b> double A[BSIZE], B[BSIZE];  ireceive A; isend CTS A;  ireceive B; isend CTS B;  for( i = 0 ; i < n ; i++ ) { wait A received; sum A; ireceive A; isend CTS A;  wait B received; sum B; ireceive B; isend CTS B; }
---	---

**Figure 1. Pseudocode for Double Buffering**

## 2 Background

Like most supercomputers, ASCI/Red and the systems software for ASCI/Red was developed to support "resource constrained applications," applications for which the problem size can be scaled to consume all of one or more of the resources provided by the computing system. That is, the size of the problem is constrained by the availability of specific resources. In many cases, these applications are constrained by the availability of processor cycles. The ability to manage processor cycles is critical for these applications.

To support application programmers in their efforts to manage processor cycles, the MPI standard includes non-blocking send and receive operations. These operations are included in the standard to permit overlap between computation and communication. In particular, an application programmer can initiate a non-blocking communication operation (either a send or a receive) and continue with a meaningful part of their computation while the communication progresses. Later, the application can poll for the completion of the communication. If the standard only provided blocking operations, the processor cycles during communication would not be available to the application and would be wasted.

### 2.1 Double Buffering

Perhaps the simplest example of overlapping computation and communication involves the use of double

buffering. In an earlier experiment[6], we measured the time taken to produce and sum a long stream of floating point numbers. In this experiment, the application consists of two processes, a *producer* and a *consumer*. The producer uses a random number generator to produce a stream of double precision floating point values and the consumer calculates the sum of the values it receives. The producer prepares a batch of numbers which are then sent to the consumer. The consumer provides two buffers so that the producer can fill one buffer while the consumer is processing the other buffer. Pseudo-code for the producer and consumer processes is presented in Figure 1.

In examining this code, notice that the producer uses non-blocking sends to transmit filled buffers. This should allow the producer to overlap its filling of one buffer with the sending of the previously filled buffer. Similarly, the consumer uses pre-posted, non-blocking receives. When the producer is faster than the consumer, this should allow the consumer to overlap the processing of one buffer with the reception of the next buffer.

Our experiments compared MPICH/GM[7] with an implementation of MPI over Portals[2]. Even though the MPICH/GM communication bandwidth was substantially higher (80 MB/s versus 50 MB/s), the overall processing rate was significantly better, 15%, when we used MPI over Portals. The improvement is due to the fact that the MPI over Portals implementation allows complete overlap of computation with communication.

## 2.2 Availability and Utilization

The reader may note that we present our discussion and results in terms of processor *availability*, how much of the processor is available to the application, rather than processor *utilization*, how much of the processor is utilized during communication. The two terms are effectively inverses of one another, that is, high availability implies low utilization and vice versa. While it is common practice to report processor utilization, we find that utilization sends the wrong message. Utilization seems to be a good thing and one naturally assumes that higher utilization is better when, in fact, the opposite is actually the case.

Ultimately, the difference is one of perspective. We find that using the term availability keeps us focused on the fact that we are trying to provide resources to applications.

## 3 The COMB Benchmark Suite

In 1999, White and Bova[10] noted that many MPI implementations do not support overlapping computation with communication, even though it is clear that the MPI standard intended that implementations support this overlap. Given our experience in the double buffering experiment and the observation of White and Bova, we set out to measure the degree to which MPI implementations supported overlap between computation and communication.

The COMB benchmark[5] suite consists of two different methods for measuring the performance of a system, each with a different perspective on characterizing the ability to overlap computation and MPI communication. This multi-method approach captures performance data on a wider range of the system and allows for results from each benchmark to be validated and/or reinforced by the other. The first method, the *Polling Method*, allows for the maximum possible overlap of computation and MPI communication. The second method, the *Post-Work-Wait Method* tests for overlap under practical restrictions on MPI calls.

### 3.1 Polling Method

The *polling method* uses two processes, one process, the *worker process*, counts cycles and performs message passing. A second, *support process*, runs on the second node and only performs message passing. Figure 2 presents pseudo code for the worker process. All receives are posted before sends. Initial setup of message

```

read current time
for( i = 0 ; i < work/poll_factor ; i++ ){
  for( j = 0 ; j < poll_factor ; j++){
    /* nothing */
  }
  if(asynchronous receive is complete){
    start asynchronous reply(s)
    post asynchronous receive(s)
  }
}
read current time

```

Figure 2. Polling Method Psuedocode For Worker Process

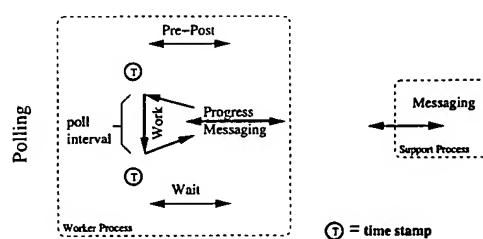


Figure 3. Overview of Polling Method

passing as well as conclusion of same are omitted from the figure. Additionally, Figure 3 provides a pictorial representation of the method.

This method uses a ping-pong communication strategy with messages flowing in both directions between sender node and receiver. Each process polls for message arrivals and propagates replacement messages upon completion of earlier messages. After a predetermined amount of computation, bandwidth and CPU availability are computed. The polling interval can be adjusted to demonstrate the trade-off between bandwidth and CPU availability. Because this method never blocks waiting for message completion it provides an accurate report of CPU availability.

As can be seen in Figure 2, after a fixed number of iterations in the inner loop the worker process polls for receipt of the next message. The number of iterations of the inner loop determines the time between polls and, hence, determines the polling interval. If a test for completion is negative, the worker process will iterate through another polling interval before testing again. If a test for completion is positive, the process will post related messaging calls and will similarly address any other received messages before entering another polling

interval. The support process sends messages as fast as they are consumed by the receiver.

We vary the polling interval to elicit changes in CPU availability and bandwidth. When the polling interval becomes sufficiently large all possible message transfers may complete during the polling interval and communication then must wait, resulting in decreased bandwidth.

The polling method uses a queue of messages at each node in order to maximize achievable bandwidth. When either process detects that a message has arrived, it iterates through the queue of all messages that have arrived, sending replies to each of these messages. When we set the queue size to one, a single message passed between the two nodes then the polling method acts as a standard ping-pong test and maximum sustained bandwidth will be sacrificed.

The benchmark actually runs in two phases. During the first, *dry run*, phase the amount of time to accomplish a predetermined amount of work in the absence of communication is recorded. The second phase records the time for the same amount of work while the two processes are exchanging messages. The CPU availability is reported as:

$$\text{availability} = \frac{\text{time( work without messaging )}}{\text{time( work plus MPI calls while messaging )}}$$

The polling method reports message passing bandwidth and CPU availability, both as functions of the polling interval.

### 3.2 Post-Work-Wait Method

While the polling method yields a great deal of useful information, it does not identify implementations that violate the “progress rule” of MPI. This rule requires communication progress even if the application is involved in computation and never makes a call to the MPI library. Because the polling method makes regular, polling calls to the MPI library, it cannot uncover problems related to the progress rule. Here it is worth noting that MPICH/GM does very well when evaluated using the polling method. The problems that we observed in the double buffering experiment are only apparent when you do not mix MPI library calls during communication.

The *Post-Work-Wait Method* mixes MPI communication and computation in a serial manner: post non-blocking MPI messages, perform computation (the work phase), and wait for the messages to complete. This strict order introduces a significant and reasonable restriction at the application level: the underlying communication system can overlap MPI communication and

computation only if, after the initial MPI calls, the message passing system requires no further intervention by the application in order to progress communication. We define the term *application offload* to describe this capability. The PWW method detects whether systems exhibit application offload and identifies where host cycles are spent on communication.

Figure 4 presents a pictorial representation of the method. With respect to communication, the PWW method performs message handling in a repeated pair of operations: 1) posting non-blocking send and receive calls and 2) wait for the messaging to complete. Both processes simultaneously send and receive a single message. The worker process performs work after the non-blocking calls before waiting for message completion. The work interval is varied to effect changes in CPU availability and bandwidth.

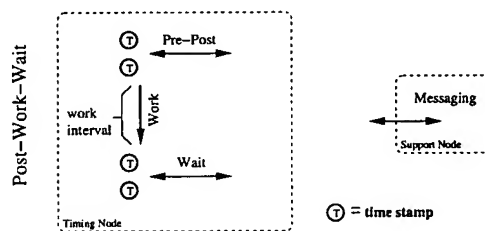


Figure 4. Post-Work-Wait Method

The PWW method collects wall clock durations for the different phases of the method. Specifically, the method collects individual durations for i) the non-blocking call phase, ii) the work phase, and iii) the wait phase. Of course, the method also records the time necessary to do the work in the absence of messaging. These phase durations are useful in identifying communication bottle necks or other causes of poor communication.

## 4 Sandia/Intel ASCI/Red Machine

The Sandia/Intel ASCI/Red machine[8] is the Department of Energy’s Accelerated Strategic Computing Initiative (ASCI) Option Red machine. It was installed at Sandia National Laboratories in 1997 and was the first computing system to demonstrate a sustained teraFLOPS level of performance. The following briefly describes the hardware and system software environment of the machine.

## 4.1 Hardware

ASCI/Red is composed of more than nine thousand 333 MHz Pentium II Xeon processors connected by a network capable of delivering 400 MB/s unidirectional communication bandwidth. Each compute node contains two processors and 256 MB of main memory. Each compute node also has a network interface chip (NIC) that resides on the memory bus, allowing for low-latency access to the network.

## 4.2 Software

The compute nodes of ASCI/Red run a variant of a lightweight kernel, called Puma [9], that was designed and developed by Sandia and the University of New Mexico. A key component of the design of Puma is a high-performance data movement layer called Portals.

Portals in Puma are data structures in an application's address space that determine how the kernel should respond to message-passing events. Portals allow the kernel to deliver messages directly to the application without any intervention by the application process. In particular, the application process need not be the currently scheduled process or perform any message selection operations, such as tag matching, to process incoming messages. We refer to this feature as *application offload*, since the application need not be involved in the transfer of data once the operation has been set up.

In Puma, all of the resources on a compute node are managed by the *system processor*. This is the only processor that performs any significant processing in supervisor mode. The remaining processor runs application code and only rarely enters supervisor mode. This processor is called the *user processor*. This arrangement produces a slight asymmetry in the performance of the processors, but it greatly simplifies the structure of the Puma kernel and maximizes the processor cycles available to the applications.

## 4.3 Processor Modes

Puma supports four different modes that allow different distributions of application processes on the processors. The processor mode is determined at run-time for the processes in a parallel job when the job is launched. The following describes each of these processor modes.

The simplest processor usage mode is to run both the kernel and application process on the system processor. This mode is commonly referred to as "heater mode"

since the second processor is not used and only generates heat. In this mode, the kernel runs only when responding to network events or in response to a system call from the application process. This mode does not offer any significant performance advantages to the application process.

In the second mode, message co-processor mode, the kernel runs on the system processor and the application process runs on the user processor. When the processors are configured in this mode, the kernel runs continuously waiting to process events from external devices or service system call requests from the application process. Because the time to transition from user mode, to supervisor mode, and back to user mode can be significant, this mode offers the advantage of reduced network latency and faster system call response time. Because of the increased message passing performance, this mode favors applications that are latency bound.

In the third mode, compute co-processor mode, the system processor and user processor both run the kernel and an application process. However, the kernel code running on the application processor does not perform any resource management activities, it simply notifies the system processor when a system call is performed. The advantage of this mode is that it provides more processor cycles for the application. However, the two processors are not symmetric since the part of the application running on the shared system processor will not progress as rapidly as the portion of the application running on the dedicated user processor. In order to use this mode, the application must use a non-standard library interface that executes a co-routine on the application processor. Because of the opportunity to utilize both processors, this mode favors applications that are compute bound.

Finally, in the fourth mode, known as virtual node mode, the system processor runs both the kernel and an application process, while the second processor also runs the kernel and a full separate application process. This mode essentially allows a compute node to be viewed by the runtime system as two independent compute nodes. The asymmetry of compute co-processor mode also exists in this mode, so the application process running on the user processor is likely to receive slightly more processor cycles than the application process running with the kernel on the system processor. This mode allows applications to avail of the user processor more easily, since the application does not need to be modified to use the non-standard co-routine interface.

In the remainder of this paper, we restrict our discussion to a comparison between standard mode (proc mode



0) and message co-processor mode (proc mode 1).

#### 4.4 MPI Implementation

The MPI library for Puma Portals on ASCI/Red [3] is a port of the MPICH [4] implementation version 1.0.12. This implementation of MPI was validated as a product by Intel for ASCI/Red in 1997 after significant testing and has been in production use with few changes since.

The performance of the MPI implementation on ASCI/Red was studied [1] using traditional ping-pong latency and bandwidth tests. In comparison to the performance of the underlying Portals layer, MPI was shown to nominally increase latency and was able to achieve nearly identical bandwidth performance in both standard processor mode and message co-processor mode. Figure 5 shows the MPI half round trip latency performance for the standard mode (proc0) and message co-processor mode (proc1). Figure 6 shows the MPI bandwidth numbers for these processor modes.

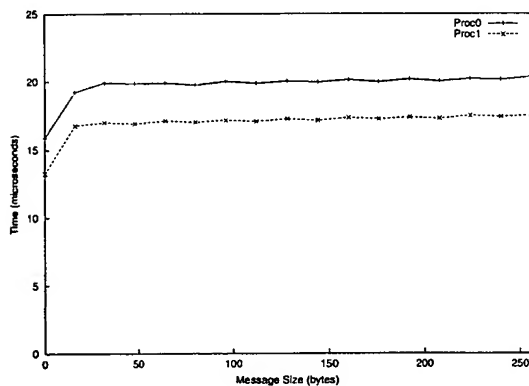


Figure 5. MPI Half Round-Trip Latency

## 5 COMB Results and Analysis

Because the results obtained using the ping-pong benchmark in 1997 did not uncover any unexpected performance issues, we turned our efforts to other projects. One of those projects involved the development of the COMB suite. The intent of the COMB suite was to evaluate the ability of MPI implementations to overlap computation with communication on high-end clusters, in particular systems built with programmable network interface cards like Myrinet or the Alteon Acenic Gigabit

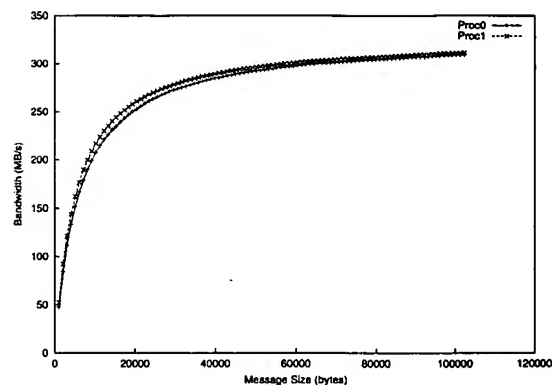


Figure 6. MPI Ping-Pong Bandwidth

Ethernet cards. On a whim, we thought it would be interesting to run the benchmarks contained in the COMB suite on ASCI/Red.

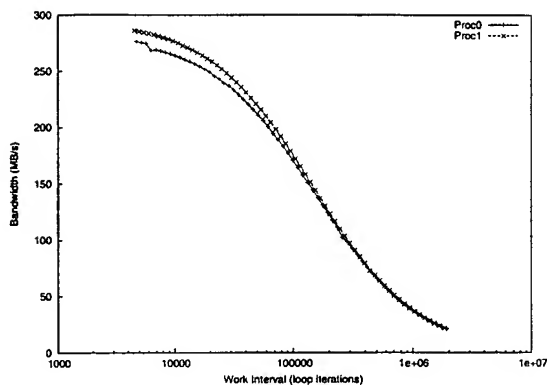
In this section we describe the performance problem that the PWW method revealed, how the MPI implementation was modified, and show the impact of this change on the performance of all of the benchmarks, including latency and bandwidth.

### 5.1 Initial Results

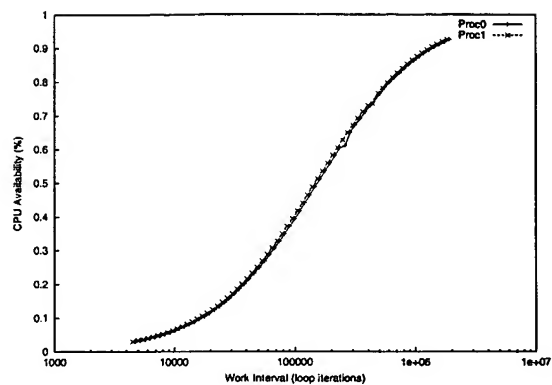
Figure 7 presents the bandwidth as a function of the work interval for 100 KB messages. This figure includes separate graphs for both processor modes. Given the differences in how bandwidth is measured in the PWW and ping-pong benchmarks, the results presented in Figure 7 are consistent with the earlier results presented in Figure 6.

In addition to bandwidth, the PWW benchmark reports the processor availability during communication. In this case, availability is reported as the ratio between the time to complete the work interval with no communication and the time to complete the work interval (and wait for message completion) while communication is progressing. Figure 8 shows CPU availability as a function of the work interval for 100 KB messages on all three processor modes.

The general shape of the curves shown in Figure 8 reflects the PWW definition of availability. When the work interval is relatively small, the work interval is too short to cover the time needed to transmit the message. This *wait while delayed* functionality suppresses apparent CPU availability until the work interval becomes sufficiently long to fill the delay period of time.



**Figure 7. MPI Bandwidth for 100KB Messages**



**Figure 8. CPU Availability for 100KB Messages**

While the shape of the curves was expected, we were surprised by the fact that there was no separation between these curves. Given that message passing is entirely handled by the system processor in message co-processor mode, we had expected that the availability would be significantly higher

Further analysis of the data provided by the PWW benchmark identified the source of the problem. In particular, the PWW benchmark provides the time taken to post each message. In message co-processor mode, we would expect that the time to post a message would have little or no dependence on the size of the message, since the application process can make a request to the kernel to start the transfer and then return to computation. However, when we looked at the posting times for various message sizes, we saw that PWW was reporting a direct relationship between the post time and the length of the message. Figure 9 shows the post time for four message sizes across varying work intervals in message co-processor mode.

The results in Figure 9 indicated that the MPI non-blocking operations were waiting for the data transfer to be completed before returning from the library. A quick inspection of the MPI implementation confirmed that non-blocking MPI send calls would trap to the kernel for the data transfer request and then immediately wait for the kernel to complete the transfer before returning from the call. By doing this, the MPI library eliminates any possibility of overlapping computation with sending messages.

This limitation does not have any effect in standard mode, since the kernel must complete the data transfer

before returning control the application process anyway. The standard ping-pong benchmark used to measure latency and bandwidth does not reveal this problem either, since it only evaluates network performance and does not consider processor overhead. For these reasons, the loss of opportunity to overlap in message co-processor mode went undetected.

## 5.2 MPI Enhancement

The MPI implementation was modified to return immediately after making a send request to the kernel. This change involved moving the structure that indicates send completion from the local stack into the send request structure. Rather than waiting for completion of the send immediately after making the request, the MPI implementation checks or waits for completion in the *MPI\_Test()* or *MPI\_Wait()* family of functions. In message co-processor mode, this gives the kernel an opportunity to transfer the data while the application process continues computing.

## 5.3 Impact of the Change

We now present several results that show the impact of this small change on latency, bandwidth, overhead, and CPU availability for the different processor modes in Puma. In some cases, this enhancement led to significant gains in performance.

Figure 10 presents the processor availability graph for using the modified MPI implementation. We now

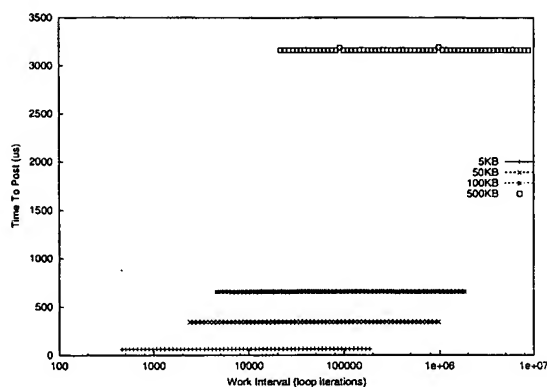


Figure 9. Time to Post in Message Co-Processor Mode

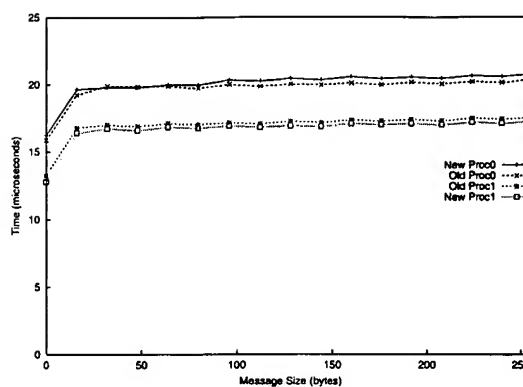


Figure 11. New MPI Half Round-Trip Latency

see the improvement in processor availability for message co-processor mode that we had expected to see.

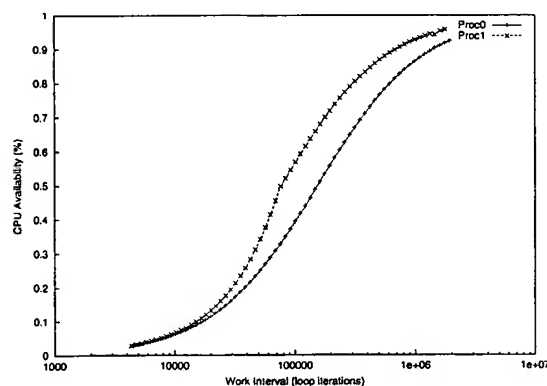


Figure 10. CPU Availability for 100KB Messages

mentation exceeds the old one at around 3 KB. In message co-processor mode, the numbers are virtually identical.

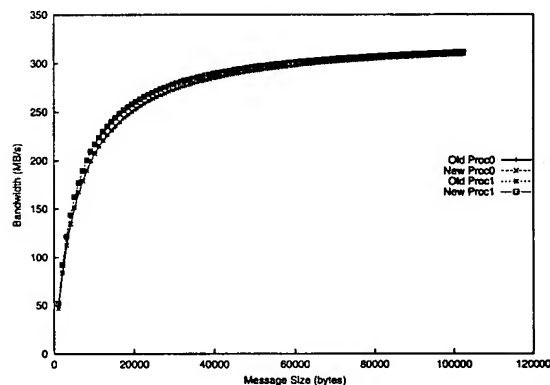


Figure 12. New MPI Ping-Pong Bandwidth

Figure 11 compares the MPI half round trip latency performance of the previous implementation with the new implementation for both processor modes. The results are mixed. For message co-processor mode, latency was improved by about 1  $\mu$ sec. However, in standard mode, the new implementation is about 1  $\mu$ sec worse.

Figure 12 compares the MPI bandwidth performance of the previous implementation with the new implementation. The numbers are nearly identical for both modes. For standard mode, the performance of the new imple-

## 6 Discussion

While the COMB benchmark suite was initially developed to compare MPI implementations for cluster systems, we thought it might be interesting to run the benchmarks on ASCI/Red. The PWW benchmark was a valuable tool that revealed a significant performance problem with the MPI implementation on ASCI/Red. We believe this benchmark to be a valuable tool in measuring message passing performance relative to proces-

sor availability. We have demonstrated its ability to expose and help diagnose performance problems that other traditional message passing benchmarks do not.

## References

- [1] R. Brightwell and D. S. Greenberg. Experiences implementing the MPI standard on sandia's lightweight kernels. Technical Report SAND97-2519, Sandia National Laboratories, August 1997.
- [2] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.
- [3] R. Brightwell and L. Shuler. Design and implementation of MPI on Puma Portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.
- [4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
- [5] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. Comb: A portable benchmark suite for assessing MPI overlap. Technical Report TR-CS-2002-13, Computer Science Department, The University of New Mexico, April 2002.
- [6] A. B. Maccabe, W. Lawry, C. Wilson, and R. Riesen. Distributing application and OS functionality to improve application performance. Technical Report TR-CS-2002-11, Computer Science Department, The University of New Mexico, April 2002.
- [7] Myricom, Inc. The GM message passing system. Technical report, Myricom, Inc., 1997.
- [8] Sandia National Laboratories. *ASCI Red*, 1996. <http://www.sandia.gov/ASCI/TFLOP>.
- [9] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.
- [10] J. B. White and S. W. Bova. Where's the overlap?: An analysis of popular mpi implementations. In *Proceedings of the Third MPI Developers' and Users' Conference*, March 1999.